WS 2016/17

Einführung in die Informatik 1

Harald Räcke

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2016WS/info1/

Wintersemester 2016/17

Diese Vorlesungsfolien basieren zum Grossteil auf der Einführungsvorlesung von Prof. Helmut Seidl (WS 2012/13).

WS 2016/17

Einführung in die Informatik 1

Harald Räcke

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2016WS/info1/

Wintersemester 2016/17

1 Vom Problem zum Program

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
E = zulässige Eingaben und A = mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- Primzahltest: $f : \mathbb{N} \to \{\text{true}, \text{false}\}$
- Schach: $f : \mathcal{E} \to \mathcal{A}$, wobei $\mathcal{E}$ die Menge aller Schachpositionen ist, und $f(e)$ der beste Zug in Position e.

1 Vom Problem zum Program

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- Primzahltest: $f : \mathbb{N} \to \{\text{true}, \text{false}\}$
- Schach: $f : \mathcal{S} \to \mathcal{Z}$, wobei $\mathcal{S}$ die Menge aller Schachpositionen ist, und $f(S) =$ der beste Zug in Position S.

1 Vom Problem zum Program

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

1 Vom Problem zum Program

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

1 Vom Problem zum Program

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit
$E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

Algorithmus

Ein Algorithmus ist ein exaktes Verfahren zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus berechnet eine Funktion f.

Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain

Abu Abdallah Muhamed ibn Musa al-Chwarizmi, ca. 780–835

1 Vom Problem zum Program

Ein Problem besteht darin, aus einer Menge von Informationen eine weitere (unbekannte) Information zu bestimmen.

mathematisch:
Ein Problem beschreibt eine Funktion $f : E \to A$, mit $E =$ zulässige Eingaben und $A =$ mögliche Ausgaben.

Beispiele:

- Addition: $f : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
- Primzahltest: $f : \mathbb{N} \to \{\text{yes}, \text{no}\}$
- Schach: $f : \mathcal{P} \to \mathcal{Z}$, wobei $\mathcal{P}$ die Menge aller Schachpositionen ist, und $f(P)$, der beste Zug in Position P.

Algorithmus

Beobachtung:
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

Beweisidee: (↑Diskrete Strukturen)

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

Algorithmus

Ein Algorithmus ist ein exaktes Verfahren zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus berechnet eine Funktion f.

Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain

Abu Abdallah Muhamed ibn Musa al-Chwarizmi, ca. 780–835

Algorithmus

Beobachtung:
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

Beweisidee: (↑Diskrete Strukturen)

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

Algorithmus

Ein Algorithmus ist ein exaktes Verfahren zur Lösung eines Problems, d.h. zur Bestimmung der gewünschten Resultate.

Man sagt auch ein Algorithmus berechnet eine Funktion f.

Ausschnitt aus Briefmarke, Soviet Union 1983
Public Domain

Abu Abdallah Muhamed ibn Musa al-Chwarizmi, ca. 780–835

Algorithmus

Das exakte Verfahren besteht i.a. darin, eine Abfolge von elementaren Einzelschritten der Verarbeitung festzulegen.

Beispiel: Alltagsalgorithmen

Resultat	*Algorithmus*	*Einzelschritte*
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

Algorithmus

Beobachtung:
Nicht jedes Problem läßt sich durch einen Algorithmus lösen (↑Berechenbarkeitstheorie).

Beweisidee: (↑Diskrete Strukturen)

- es gibt überabzählbar unendlich viele Probleme
- es gibt abzählbar unendlich viele Algorithmen

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\mathrm{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Algorithmus

Das exakte Verfahren besteht i.a. darin, eine Abfolge von elementaren Einzelschritten der Verarbeitung festzulegen.

Beispiel: Alltagsalgorithmen

Resultat	*Algorithmus*	*Einzelschritte*
Pullover	Strickmuster	eine links, eine rechts, eine fallen lassen
Kuchen	Rezept	nimm 3 Eier ...
Konzert	Partitur	Noten

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

Beispiel: Euklidischer Algorithmus

Problem: geg. $a,b \in \mathbb{N}, a,b \neq 0$. Bestimme $\mathrm{ggT}(a,b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\mathrm{ggT}(a,b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Beispiel: Euklidischer Algorithmus

Problem: geg. $a,b \in \mathbb{N}, a,b \neq 0$. Bestimme $\mathrm{ggT}(a,b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\mathrm{ggT}(a,b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\mathrm{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a, b) = \mathrm{ggT}(a - b, b)$.

Seien $g = \mathrm{ggT}(a, b)$, $g' = \mathrm{ggT}(a - b, b)$.
Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a - b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

$$\begin{aligned} a - b &= (q_a - q_b) \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a &= (q'_{a-b} + q'_b) \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

Beispiel: Euklidischer Algorithmus

Problem: geg. $a, b \in \mathbb{N}, a, b \neq 0$. Bestimme $\mathrm{ggT}(a, b)$.

Algorithmus:

1. Falls $a = b$ brich Berechnung ab. Es gilt $\mathrm{ggT}(a, b) = a$. Ansonsten gehe zu Schritt 2.
2. Falls $a > b$, ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Ansonsten gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a, b) = \mathrm{ggT}(a - b, b)$.

Seien $g = \mathrm{ggT}(a, b)$, $g' = \mathrm{ggT}(a - b, b)$.
Dann gilt:

$$\begin{aligned} a &= q_a \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a - b &= q'_{a-b} \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

$$\begin{aligned} a - b &= (q_a - q_b) \cdot g \\ b &= q_b \cdot g \end{aligned} \quad \text{und} \quad \begin{aligned} a &= (q'_{a-b} + q'_b) \cdot g' \\ b &= q'_b \cdot g' \end{aligned}$$

Das heißt g ist Teiler von $a - b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \leq g'$ und $g' \leq g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

$$\begin{array}{rcl} a-b & = & (q_a - q_b) \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a & = & (q'_{a-b} + q'_b) \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a-b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \le g'$ und $g' \le g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

$$\begin{array}{rcl} a-b & = & (q_a - q_b) \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a & = & (q'_{a-b} + q'_b) \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a-b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \le g'$ und $g' \le g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

$$\begin{array}{rcl} a-b & = & (q_a - q_b) \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a & = & (q'_{a-b} + q'_b) \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a-b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \le g'$ und $g' \le g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

$$\begin{array}{rcl} a-b & = & (q_a - q_b) \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a & = & (q'_{a-b} + q'_b) \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a-b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \le g'$ und $g' \le g$, also $g = g'$.

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Beispiel: Euklidischer Algorithmus

Warum geht das?
Wir zeigen, für $a > b$: $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Seien $g = \mathrm{ggT}(a,b)$, $g' = \mathrm{ggT}(a-b,b)$.
Dann gilt:

$$\begin{array}{rcl} a & = & q_a \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a-b & = & q'_{a-b} \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

$$\begin{array}{rcl} a-b & = & (q_a - q_b) \cdot g \\ b & = & q_b \cdot g \end{array} \quad \text{und} \quad \begin{array}{rcl} a & = & (q'_{a-b} + q'_b) \cdot g' \\ b & = & q'_b \cdot g' \end{array}$$

Das heißt g ist Teiler von $a-b, b$ und g' ist Teiler von a, b.

Daraus folgt $g \le g'$ und $g' \le g$, also $g = g'$.

Programm

Ein Programm ist die Formulierung eines Algorithmus in einer Programmiersprache.

Die Formulierung gestattet (hoffentlich) eine maschinelle Ausführung.

- Ein Programmsystem berechnet i.a. nicht nur eine Funktion, sondern immer wieder Funktionen in Interaktion mit Benutzerinnen und/oder der Umgebung.
- Es gibt viele Programmmiersprachen: Java, C, Prolog, Fortran, TeX, PostScript, ...

Eigenschaften

(statische) Finitheit. Die Beschreibung des Algorithmus besitzt endliche Länge. (↑nichtuniforme Algorithmen)

(dynamische) Finitheit. Die bei Abarbeitung entstehenden Zwischenergebnisse sind endlich.

Terminiertheit. Algorithmen, die nach endlich vielen Schritten ein Resultat liefern, heißen terminierend. (↑Betriebssysteme, ↑reaktive Systeme)

Determiniertheit. Bei gleichen Eingabedaten gibt ein Algorithmus das gleiche Ergebnis aus. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Determinismus. Der nächste anzuwendende Schritt im Verfahren ist stets eindeutig definiert. (↑randomisierte Algorithmen, ↑nicht-deterministische Algorithmen)

Programm

Ein Programm ist gut, wenn

- die Programmiererin in ihr algorithmische Ideen natürlich beschreiben kann, insbesondere später noch versteht was das Programm tut (oder nicht tut);
- ein Computer das Programm leicht verstehen und effizient ausführen kann.

Programm

Ein Programm ist die Formulierung eines Algorithmus in einer Programmiersprache.

Die Formulierung gestattet (hoffentlich) eine maschinelle Ausführung.

- Ein Programmsystem berechnet i.a. nicht nur eine Funktion, sondern immer wieder Funktionen in Interaktion mit Benutzerinnen und/oder der Umgebung.
- Es gibt viele Programmmiersprachen: Java, C, Prolog, Fortran, TeX, PostScript, ...

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- Datenstrukturen anbieten
- Operationen auf Daten erlauben
- Kontrollstrukturen zur Ablaufsteuerung bereitstellen

Als Beispiel betrachten wir MiniJava.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in MiniJava zu nutzen müssen sie zunächst eingeführt, d.h. deklariert werden.

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- Datenstrukturen anbieten
- Operationen auf Daten erlauben
- Kontrollstrukturen zur Ablaufsteuerung bereitstellen

Als Beispiel betrachten wir MiniJava.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in MiniJava zu nutzen müssen sie zunächst eingeführt, d.h. deklariert werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in MiniJava zu nutzen müssen sie zunächst eingeführt, d.h. deklariert werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in MiniJava zu nutzen müssen sie zunächst eingeführt, d.h. deklariert werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in MiniJava zu nutzen müssen sie zunächst eingeführt, d.h. deklariert werden.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Variablen

Variablen dienen zur Speicherung von Daten.

Um Variablen in MiniJava zu nutzen müssen sie zunächst eingeführt, d.h. deklariert werden.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die Zuweisung.

Beispiele:

- `x = 7;`
  Die Variable `x` erhält den Wert 7.
- `result = x;`
  Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- `result = x + 19;`
  Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die Zuweisung.

Beispiele:

- `x = 7;`
  Die Variable `x` erhält den Wert 7.
- `result = x;`
  Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- `result = x + 19;`
  Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die Zuweisung.

Beispiele:

- `x = 7;`
  Die Variable `x` erhält den Wert 7.
- `result = x;`
  Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- `result = x + 19;`
  Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Variablen

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den Namen `x` und `result` ein.

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen („Integers“) speichern sollen.

  `int` heißt auch Typ der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas „,“ getrennt.
- Am Ende steht ein Semikolon „;“.

Operationen – Zuweisung

Achtung:

- Java bezeichnet die Zuweisung mit „=“ anstatt „:=“ (Erbschaft von C...)
- Eine Zuweisung wird mit „;“ beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert vor der Zuweisung zu.

Operationen – Zuweisung

Operationen gestatten es, Werte von Variablen zu ändern. Die wichtigste Operation ist die Zuweisung.

Beispiele:

- `x = 7;`
  Die Variable `x` erhält den Wert 7.
- `result = x;`
  Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- `result = x + 19;`
  Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

Operationen – Input/Output

MiniJava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- `x = read();`
  Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- `write(42);`
  Schreibt `42` auf die Ausgabe.
- `write(result);`
  Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
  Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- Java bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von C...)
- Eine Zuweisung wird mit „`;`“ beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert vor der Zuweisung zu.

Operationen – Input/Output

MiniJava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- `x = read();`
  Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- `write(42);`
  Schreibt `42` auf die Ausgabe.
- `write(result);`
  Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
  Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- Java bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von C...)
- Eine Zuweisung wird mit „`;`“ beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert vor der Zuweisung zu.

Operationen – Input/Output

MiniJava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- `x = read();`
  Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- `write(42);`
  Schreibt `42` auf die Ausgabe.
- `write(result);`
  Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
  Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- Java bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von C...)
- Eine Zuweisung wird mit „`;`“ beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert vor der Zuweisung zu.

Operationen – Input/Output

MiniJava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- `x = read();`
  Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- `write(42);`
  Schreibt `42` auf die Ausgabe.
- `write(result);`
  Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
  Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Zuweisung

Achtung:

- Java bezeichnet die Zuweisung mit „`=`“ anstatt „`:=`“ (Erbschaft von C...)
- Eine Zuweisung wird mit „`;`“ beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert vor der Zuweisung zu.

Operationen – Input/Output

Achtung:

- Das argument der `write`-Operation in den Beispielen ist ein `int`.
- Um es ausgeben zu können muss es erst in ein Zeichenfolge umgewandelt werden, d.h. einen `String`

In MiniJava könne auch direkt Strings ausgegeben werden:

Beispiel:

- `write("Hello World!!!");`
  Schreibt `Hello World!!!` auf die Ausgabe.

Operationen – Input/Output

MiniJava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- `x = read();`
  Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- `write(42);`
  Schreibt `42` auf die Ausgabe.
- `write(result);`
  Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
  Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Operationen – Input/Output

Achtung:

- Das argument der `write`-Operation in den Beispielen ist ein `int`.
- Um es ausgeben zu können muss es erst in ein Zeichenfolge umgewandelt werden, d.h. einen `String`

In MiniJava könne auch direkt Strings ausgegeben werden:

Beispiel:

- `write("Hello World!!!");`
  Schreibt `Hello World!!!` auf die Ausgabe.

Operationen – Input/Output

MiniJava enthält Operationen um Daten (Zahlen) einlesen bzw. ausgeben zu können.

Beispiele:

- `x = read();`
  Liest eine Folge von Zeichen ein und interpretiert sie als ganze Zahl, deren Wert sie der Variablen `x` als Wert zuweist.
- `write(42);`
  Schreibt `42` auf die Ausgabe.
- `write(result);`
  Bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
  Bestimmt den Wert der Variablen `x`, subtrahiert `14` und schreibt das Ergebnis auf die Ausgabe.

Kontrollstrukturen – Sequenz

Sequenz:

```
1 int x, y, result;
2 x = read();
3 y = read();
4 result = x + y;
5 write(result);
```

- Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- Jede Operation wird genau einmal ausgeführt.
- Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen.
- Mit Beendigung der letzten Operation endet die Programm-Ausführung.

Sequenz alleine erlaubt nur sehr einfache Programme.

Operationen – Input/Output

Achtung:

- Das argument der `write`-Operation in den Beispielen ist ein `int`.
- Um es ausgeben zu können muss es erst in ein Zeichenfolge umgewandelt werden, d.h. einen `String`

In MiniJava könne auch direkt Strings ausgegeben werden:

Beispiel:

- `write("Hello World!!!");`
  Schreibt `Hello World!!!` auf die Ausgabe.

Kontrollstrukturen – Selektion

Selektion (bedingte Auswahl):

```
1 int x, y, result;
2 x = read();
3 y = read();
4 if (x > y)
5     result = x - y;
6 else
7     result = y - x;
8 write(result);
```

- Zuerst wird die Bedingung ausgewertet
- Ist sie erfüllt, wird die nächste Operation ausgeführt.
- Ist sie nicht erfüllt, wird die nächste Operation nach dem `else`-Zweig ausgeführt.

Kontrollstrukturen – Sequenz

Sequenz:

```
1 int x, y, result;
2 x = read();
3 y = read();
4 result = x + y;
5 write(result);
```

- Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- Jede Operation wird genau einmal ausgeführt.
- Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen.
- Mit Beendigung der letzten Operation endet die Programm-Ausführung.

Sequenz alleine erlaubt nur sehr einfache Programme.

Kontrollstrukturen – Selektion

Beispiel:

- Statt einer einzelnen Operation können die Alternativen auch aus Statements bestehen:

```
1 int x;
2 x = read();
3 if (x == 0)
4     write(0);
5 else if (x < 0)
6     write(-1);
7 else
8     write(+1);
```

Kontrollstrukturen – Selektion

Selektion (bedingte Auswahl):

```
1 int x, y, result;
2 x = read();
3 y = read();
4 if (x > y)
5     result = x - y;
6 else
7     result = y - x;
8 write(result);
```

- Zuerst wird die Bedingung ausgewertet
- Ist sie erfüllt, wird die nächste Operation ausgeführt.
- Ist sie nicht erfüllt, wird die nächste Operation nach dem `else`-Zweig ausgeführt.

Kontrollstrukturen – Selektion

Beispiel:

- . . . oder aus (geklammerten) Folgen von Operationen und Statements:

```
1  int x, y;
2  x = read();
3  if (x != 0) {
4      y = read();
5      if (x > y)
6          write(x);
7      else
8          write(y);
9  } else
10     write(0);
```

Kontrollstrukturen – Selektion

Beispiel:

- Statt einer einzelnen Operation können die Alternativen auch aus Statements bestehen:

```
1  int x;
2  x = read();
3  if (x == 0)
4      write(0);
5  else if (x < 0)
6      write(-1);
7  else
8      write(+1);
```

Kontrollstrukturen – Selektion

Beispiel:

- . . . eventuell fehlt auch der `else`-Teil:

```
1 int x, y;
2 x = read();
3 if (x != 0) {
4     y = read();
5     if (x > y)
6         write(x);
7     else
8         write(y);
9 }
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden. . .

Kontrollstrukturen – Selektion

Beispiel:

- . . . oder aus (geklammerten) Folgen von Operationen und Statements:

```
1 int x, y;
2 x = read();
3 if (x != 0) {
4     y = read();
5     if (x > y)
6         write(x);
7     else
8         write(y);
9 } else
10    write(0);
```

Kontrollstrukturen – Iteration

Iteration (wiederholte Ausführung)

```
1 int x, y;
2 x = read(); y = read();
3 while (x != y) {
4     if (x < y)
5         y = y - x;
6     else
7         x = x - y;
8 }
9 write(x);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird der Rumpf des `while`-statements ausgeführt.
- Nach Ausführung des Rumpfs wird das gesamte `while`-statement erneut ausgeführt.
- Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem `while`-statement fort.

Kontrollstrukturen – Selektion

Beispiel:

- ...eventuell fehlt auch der `else`-Teil:

```
1 int x, y;
2 x = read();
3 if (x != 0) {
4     y = read();
5     if (x > y)
6         write(x);
7     else
8         write(y);
9 }
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden...

Theorem (↑Berechenbarkeitstheorie)
Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines MiniJava-Programms berechnen.

Beweisidee

- Was heißt berechenbar?
  Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.
- Schreibe ein MiniJava-Programm, das eine Turingmaschine simuliert.

Kontrollstrukturen – Iteration

Iteration (wiederholte Ausführung)

```
1 int x, y;
2 x = read(); y = read();
3 while (x != y) {
4     if (x < y)
5         y = y - x;
6     else
7         x = x - y;
8 }
9 write(x);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird der Rumpf des `while`-statements ausgeführt.
- Nach Ausführung des Rumpfs wird das gesamte `while`-statement erneut ausgeführt.
- Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem `while`-statement fort.

Theorem (↑Berechenbarkeitstheorie)
Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines MiniJava-Programms berechnen.

Beweisidee

- Was heißt berechenbar?
  Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.
- Schreibe ein MiniJava-Programm, das eine Turingmaschine simuliert.

Kontrollstrukturen – Iteration

Iteration (wiederholte Ausführung)

```
1 int x, y;
2 x = read(); y = read();
3 while (x != y) {
4     if (x < y)
5         y = y - x;
6     else
7         x = x - y;
8 }
9 write(x);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird der Rumpf des `while`-statements ausgeführt.
- Nach Ausführung des Rumpfs wird das gesamte `while`-statement erneut ausgeführt.
- Ist die Bedingung nicht erfüllt fährt die Programmausführung hinter dem `while`-statement fort.

2 Eine einfache Programmiersprache

MiniJava-Programme sind ausführbares Java. Man muss sie nur geeignet dekorieren.

Beispiel: das GGT-Programm.

```
1  int x, y;
2  x = read();
3  y = read();
4  while (x != y) {
5      if (x < y)
6          y = y - x;
7      else
8          x = x - y;
9  }
10 write(x);
```

2 Eine einfache Programmiersprache

Theorem (↑Berechenbarkeitstheorie)
Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, und Iteration, d.h., mithilfe eines MiniJava-Programms berechnen.

Beweisidee

- Was heißt berechenbar?
  Eine Funktion heißt berechenbar wenn man sie mithilfe einer Turingmaschine berechnen kann.
- Schreibe ein MiniJava-Programm, das eine Turingmaschine simuliert.

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

Daraus wird folgendes Java-Programm:

```
1  public class GGT extends MiniJava {
2      public static void main (String[] args) {
3          int x, y;
4          x = read();
5          y = read();
6          while (x != y) {
7              if (x < y)
8                  y = y - x;
9              else
10                 x = x - y;
11         }
12         write(x);
13     } // Ende der Definition von main();
14 } // Ende der Definition der Klasse GGT;
```

"GGT.java"

```java
import javax.swing.JOptionPane;
import javax.swing.JFrame;
public class MiniJava {
    public static int read () {
        JFrame f = new JFrame ();
        String s = JOptionPane.showInputDialog (f, "Eingabe:");
        int x = 0; f.dispose ();
        if (s == null) System.exit (0);
        try { x = Integer.parseInt (s.trim ());
        } catch (NumberFormatException e) { x = read (); }
        return x;
    }
    public static void write (String x) {
        JFrame f = new JFrame ();
        JOptionPane.showMessageDialog (f, x, "Ausgabe",
            JOptionPane.PLAIN_MESSAGE);
        f.dispose ();
    }
    public static void write (int x) { write (""+x); }
}
```

Datei: "MiniJava.java"

Weitere Erläuterungen:

- Jedes Programm sollte Kommentare enthalten, damit man sich selbst später noch darin zurecht findet!
- Ein Kommentar in Java hat etwa die Form:
  `// Das ist ein Kommentar!!!`
- Wenn er sich über mehrere Zeilen erstrecken soll dann

```
/* Dieser Kommentar ist verdammt
   laaaaaaaaaaaang
        */
```

```java
import javax.swing.JOptionPane;
import javax.swing.JFrame;
public class MiniJava {
    public static int read () {
        JFrame f = new JFrame ();
        String s = JOptionPane.showInputDialog (f, "Eingabe:");
        int x = 0; f.dispose ();
        if (s == null) System.exit (0);
        try { x = Integer.parseInt (s.trim ());
        } catch (NumberFormatException e) { x = read (); }
        return x;
    }
    public static void write (String x) {
        JFrame f = new JFrame ();
        JOptionPane.showMessageDialog (f, x, "Ausgabe",
            JOptionPane.PLAIN_MESSAGE);
        f.dispose ();
    }
    public static void write (int x) { write (""+x); }
}
```

Datei: "MiniJava.java"

2 Eine einfache Programmiersprache

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
raecke> javac GGT.java
raecke> java GGT
```

- Der Compiler javac liest das Programm aus den Dateien `GGT.java` und `MiniJava.java` ein und erzeugt für sie JVM-Code, den er in den Dateien `GGT.class` und `MiniJava.class` ablegt.
- Das Laufzeitsystem java liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

Weitere Erläuterungen:

- Jedes Programm sollte Kommentare enthalten, damit man sich selbst später noch darin zurecht findet!
- Ein Kommentar in Java hat etwa die Form:
```
// Das ist ein Kommentar!!!
```
- Wenn er sich über mehrere Zeilen erstrecken soll dann
```
/* Dieser Kommentar ist verdammt
   laaaaaaaaaaaang
      */
```

Ausblick

MiniJava ist sehr primitiv

Die Programmiersprache Java bietet noch eine Fülle von Hilfsmitteln an, die das Programmieren erleichtern sollen.

Insbesondere gibt es

- viele weitere Datentypen (nicht nur `int`) und
- viele weitere Kontrollstrukturen

. . . kommt später in der Vorlesung!

2 Eine einfache Programmiersprache

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
raecke> javac GGT.java
raecke> java GGT
```

- Der Compiler javac liest das Programm aus den Dateien `GGT.java` und `MiniJava.java` ein und erzeugt für sie JVM-Code, den er in den Dateien `GGT.class` und `MiniJava.class` ablegt.
- Das Laufzeitsystem java liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

3 Syntax von Programmiersprachen

Syntax („Lehre vom Satzbau“)

- formale Beschreibung des Aufbaus der „Worte“ und „Sätze“, die zu einer Sprache gehören;
- im Falle einer Programmiersprache Festlegung, wie Programme aussehen müssen.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahmelisten;
- Sprachgefühl.

3 Syntax von Programmiersprachen

Syntax („Lehre vom Satzbau“)

- formale Beschreibung des Aufbaus der „Worte“ und „Sätze“, die zu einer Sprache gehören;
- im Falle einer Programmiersprache Festlegung, wie Programme aussehen müssen.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- Listen von Schlüsselworten wie `if`, `int`, `else`, `while`...
- Regeln, wie einzelne Worte (Tokens) z.B. Namen gebildet werden.

  Frage: Ist x10 ein zulässiger Name für eine Variable (oder _ab oder A#B oder 0A?B)?...
- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

  Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- Kontextbedingungen.

  Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahmelisten;
- Sprachgefühl.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- Listen von Schlüsselworten wie `if`, `int`, `else`, `while`...
- Regeln, wie einzelne Worte (Tokens) z.B. Namen gebildet werden.

  Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

  Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- Kontextbedingungen.

  Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahmelisten;
- Sprachgefühl.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- Listen von Schlüsselworten wie `if`, `int`, `else`, `while`...
- Regeln, wie einzelne Worte (Tokens) z.B. Namen gebildet werden.

  Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

  Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- Kontextbedingungen.

  Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahmelisten;
- Sprachgefühl.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- Listen von Schlüsselworten wie `if`, `int`, `else`, `while`...
- Regeln, wie einzelne Worte (Tokens) z.B. Namen gebildet werden.

  Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

  Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- Kontextbedingungen.

  Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Hilfsmittel

Hilfsmittel bei natürlicher Sprache

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahmelisten;
- Sprachgefühl.

Beobachtung

Programmiersprachen sind

- formalisierter als natürliche Sprache
- besser für maschinelle Verarbeitung geeignet.

Hilfsmittel

Hilfsmittel bei Programmiersprachen

- Listen von Schlüsselworten wie `if`, `int`, `else`, `while`...
- Regeln, wie einzelne Worte (Tokens) z.B. Namen gebildet werden.

  Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

  Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- Kontextbedingungen.

  Beispiel: Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Syntax vs. Semantik

Semantik („Lehre von der Bedeutung“)

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine Bedeutung, d.h teilt einem Hörer/Leser einen Sachverhalt mit (↑Information)
- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine Bedeutung...

Beobachtung

Programmiersprachen sind

- formalisierter als natürliche Sprache
- besser für maschinelle Verarbeitung geeignet.

Syntax vs. Semantik

Die Bedeutung eines Programms ist

- alle möglichen Ausführungen der beschriebenen Berechnung (↑operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben (↑denotationelle Semantik).

Achtung!

Ist ein Programm syntaktisch korrekt, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. semantisch korrekt ist!!!

Syntax vs. Semantik

Semantik („Lehre von der Bedeutung“)

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine Bedeutung, d.h teilt einem Hörer/Leser einen Sachverhalt mit (↑Information)
- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine Bedeutung...

Syntax vs. Semantik

Die Bedeutung eines Programms ist

- alle möglichen Ausführungen der beschriebenen Berechnung (↑operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben (↑denotationelle Semantik).

Achtung!

Ist ein Programm syntaktisch korrekt, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. semantisch korrekt ist!!!

Syntax vs. Semantik

Semantik („Lehre von der Bedeutung“)

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine Bedeutung, d.h teilt einem Hörer/Leser einen Sachverhalt mit (↑Information)
- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine Bedeutung...

Syntax vs. Semantik

Die Bedeutung eines Programms ist

- alle möglichen Ausführungen der beschriebenen Berechnung (↑operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben (↑denotationelle Semantik).

Achtung!

Ist ein Programm syntaktisch korrekt, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. semantisch korrekt ist!!!

Syntax vs. Semantik

Semantik („Lehre von der Bedeutung“)

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine Bedeutung, d.h teilt einem Hörer/Leser einen Sachverhalt mit (↑Information)
- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine Bedeutung...

Syntax vs. Semantik

Die Bedeutung eines Programms ist

- alle möglichen Ausführungen der beschriebenen Berechnung (↑operationelle Semantik); oder
- die definierte Abbildung der Eingaben auf die Ausgaben (↑denotationelle Semantik).

Achtung!

Ist ein Programm syntaktisch korrekt, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. semantisch korrekt ist!!!

3.1 Reservierte Wörter

- `int`
  ⟹ Bezeichner für Basistypen;
- `if, else, then, while...`
  ⟹ Schlüsselwörter für Programmkonstrukte;
- `(,), ",', {,}, ,,;`
  ⟹ Sonderzeichen;

3.1 Was ist ein erlaubter Name?

- `int`
  $\Longrightarrow$ Bezeichner für Basistypen;
- `if, else, then, while...`
  $\Longrightarrow$ Schlüsselwörter für Programmkonstrukte;
- `(,), ",', {,}, ,,;`
  $\Longrightarrow$ Sonderzeichen;

3.2 Was ist ein erlaubter Name?

Schritt 1:
Festlegung erlaubter Zeichen:

letter ::= $ | _ | a | ... | z | A | ... | Z
digit ::= 0 | ... | 9

- letter und digit bezeichnen Zeichenklassen, d.h. Mengen von Zeichen, die gleich behandelt werden.
- Das Symbol „|“ trennt zulässige Alternativen.
- Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

3.1 Was ist ein erlaubter Name?

- `int`
  $\Longrightarrow$ Bezeichner für Basistypen;
- `if, else, then, while...`
  $\Longrightarrow$ Schlüsselwörter für Programmkonstrukte;
- `(,), ",', {,}, ,,;`
  $\Longrightarrow$ Sonderzeichen;

3.2 Was ist ein erlaubter Name?

Schritt 1:
Festlegung erlaubter Zeichen:

letter ::= $ | _ | a | ... | z | A | ... | Z
digit ::= 0 | ... | 9

- letter und digit bezeichnen Zeichenklassen, d.h. Mengen von Zeichen, die gleich behandelt werden.
- Das Symbol „|“ trennt zulässige Alternativen.
- Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

3.2 Was ist ein erlaubter Name?

Schritt 2:
Festlegung der Zeichenanordnung:

name ::= letter (letter | digit)*

- Erst kommt ein Zeichen der Klasse letter, dann eine (eventuell auch leere) Folge von Zeichen entweder aus letter oder aus digit.
- Der Operator „*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- Der Operator „*“ ist ein Postfix-Operator. Das heißt, er steht hinter seinem Argument.

3.2 Was ist ein erlaubter Name?

Schritt 1:
Festlegung erlaubter Zeichen:

letter ::= $ | _ | a | ... | z | A | ... | Z
digit ::= 0 | ... | 9

- letter und digit bezeichnen Zeichenklassen, d.h. Mengen von Zeichen, die gleich behandelt werden.
- Das Symbol „|“ trennt zulässige Alternativen.
- Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.

Beispiele

`_178`	`5ABC`
`Das_ist_kein_Name`	`!Hallo!`
`x`	`x'`
`_`	`a=b`
`$Password$`	`-178`
...sind legale Namen.	...sind keine legalen Namen.

Achtung
Reservierte Wörter sind als Namen verboten.

3.2 Was ist ein erlaubter Name?

Schritt 2:
Festlegung der Zeichenanordnung:

$$\text{name} \quad ::= \quad \text{letter (letter | digit)*}$$

- Erst kommt ein Zeichen der Klasse letter, dann eine (eventuell auch leere) Folge von Zeichen entweder aus letter oder aus digit.
- Der Operator „*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- Der Operator „*“ ist ein Postfix-Operator. Das heißt, er steht hinter seinem Argument.

Beispiele

_178	5ABC
Das_ist_kein_Name	!Hallo!
x	x’
_	a=b
$Password$	-178
...sind legale Namen.	...sind keine legalen Namen.

Achtung
Reservierte Wörter sind als Namen verboten.

3.2 Was ist ein erlaubter Name?

Schritt 2:
Festlegung der Zeichenanordnung:

name ::= letter (letter | digit)*

- Erst kommt ein Zeichen der Klasse letter, dann eine (eventuell auch leere) Folge von Zeichen entweder aus letter oder aus digit.
- Der Operator „*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- Der Operator „*“ ist ein Postfix-Operator. Das heißt, er steht hinter seinem Argument.

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen Konstanten.
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

number ::= digit digit*

- Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

Beispiele

`_178`	`5ABC`
`Das_ist_kein_Name`	`!Hallo!`
`x`	`x'`
`_`	`a=b`
`$Password$`	`-178`
...sind legale Namen.	...sind keine legalen Namen.

Achtung
Reservierte Wörter sind als Namen verboten.

Beispiele

`17`	
`12490`	
`42`	
`0`	`"Hello World!"`
`00070`	`0.5e+128`
...sind `int`-Konstanten	...sind keine `int`-Konstanten

3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen Konstanten.
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

$$\text{number} \quad ::= \quad \text{digit digit}^*$$

- Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen reguläre Ausdrücke (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

```
17
12490
42
0
00070
```

...sind `int`-Konstanten

```
"Hello World!"
0.5e+128
```

...sind keine `int`-Konstanten

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  $\Longrightarrow$ alle Wörter gerader Länge (über $\$, \_, a, \ldots, z, A, \ldots, Z$);
- letter* test letter*
  $\Longrightarrow$ alle Wörter, die das Teilwort test enthalten;
- _ digit* 17
  $\Longrightarrow$ alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- exp ::= (e|E)(+|-)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  $\Longrightarrow$ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen reguläre Ausdrücke (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  $\Longrightarrow$ alle Wörter gerader Länge (über $, _, a, \ldots, z, A, \ldots, Z$);
- letter* test letter*
  $\Longrightarrow$ alle Wörter, die das Teilwort test enthalten;
- _ digit* 17
  $\Longrightarrow$ alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- exp ::= (e|E)(+|-)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  $\Longrightarrow$ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen reguläre Ausdrücke (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  $\Longrightarrow$ alle Wörter gerader Länge (über $, _, a, \ldots, z, A, \ldots, Z$);
- letter* test letter*
  $\Longrightarrow$ alle Wörter, die das Teilwort test enthalten;
- _ digit* 17
  $\Longrightarrow$ alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- exp ::= (e|E)(+|-)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  $\Longrightarrow$ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen reguläre Ausdrücke (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  $\Longrightarrow$ alle Wörter gerader Länge (über $, _, a, \ldots, z, A, \ldots, Z$);
- letter* test letter*
  $\Longrightarrow$ alle Wörter, die das Teilwort test enthalten;
- _ digit* 17
  $\Longrightarrow$ alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- exp ::= (e|E)(+|-)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  $\Longrightarrow$ alle Gleitkommazahlen...

Reguläre Ausdrücke

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- * (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen reguläre Ausdrücke (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Programmverarbeitung

1. Phase (↑Scanner)

Identifizierung von

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- Whitespace,
- Kommentaren

$\Longrightarrow$ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

2. Phase (↑Parser)

Analyse der Struktur des Programms.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  $\Longrightarrow$ alle Wörter gerader Länge (über $, _, a, \ldots, z, A, \ldots, Z$);
- letter* test letter*
  $\Longrightarrow$ alle Wörter, die das Teilwort test enthalten;
- _ digit* 17
  $\Longrightarrow$ alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- exp ::= (e|E)(+|-)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  $\Longrightarrow$ alle Gleitkommazahlen...

Programmverarbeitung

1. Phase (↑Scanner)

Identifizierung von

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- Whitespace,
- Kommentaren

$\Longrightarrow$ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

2. Phase (↑Parser)

Analyse der Struktur des Programms.

Beispiele

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  $\Longrightarrow$ alle Wörter gerader Länge (über $, _, a, \ldots, z, A, \ldots, Z$);
- letter* test letter*
  $\Longrightarrow$ alle Wörter, die das Teilwort test enthalten;
- _ digit* 17
  $\Longrightarrow$ alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- exp ::= (e|E)(+|-)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  $\Longrightarrow$ alle Gleitkommazahlen...

3.4 Struktur von Programmen

Programme sind hierarchisch aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program   ::=   decl* stmt*
decl      ::=   type name ( , name )* ;
type      ::=   int
```

- Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- Eine Deklaration gibt den Typ an, hier: `int`, gefolgt von einer Komma-separierten Liste von Variablennamen.

Programmverarbeitung

1. Phase (↑Scanner)

Identifizierung von

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- Whitespace,
- Kommentaren

$\Longrightarrow$ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

2. Phase (↑Parser)

Analyse der Struktur des Programms.

3.4 Struktur von Programmen

Programme sind hierarchisch aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program   ::=   decl* stmt*
decl      ::=   type name ( , name )* ;
type      ::=   int
```

- Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- Eine Deklaration gibt den Typ an, hier: `int`, gefolgt von einer Komma-separierten Liste von Variablennamen.

Programmverarbeitung

1. Phase (↑Scanner)

Identifizierung von

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- Whitespace,
- Kommentaren

$\Longrightarrow$ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

2. Phase (↑Parser)

Analyse der Struktur des Programms.

3.4 Struktur von Programmen

Programme sind hierarchisch aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program   ::=   decl* stmt*
decl      ::=   type name ( , name )* ;
type      ::=   int
```

- Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- Eine Deklaration gibt den Typ an, hier: `int`, gefolgt von einer Komma-separierten Liste von Variablennamen.

Anweisungen

```
stmt    ::=   ; | { stmt* } |
              name = expr; | name = read(); |
              write( expr ); |
              if ( cond ) stmt |
              if ( cond ) stmt else stmt |
              while ( cond ) stmt
```

- Ein Statement ist entweder „leer“ (d.h. gleich `;`) oder eine geklammerte Folge von Statements;
- oder eine Zuweisung, eine Lese- oder Schreiboperation;
- eine (einseitige oder zweiseitige) bedingte Verzweigung;
- oder eine Schleife.

Anweisungen

```
stmt    ::=   ; | { stmt* } |
              name = expr; | name = read(); |
              write( expr ); |
              if ( cond ) stmt |
              if ( cond ) stmt else stmt |
              while ( cond ) stmt
```

- Ein Statement ist entweder „leer“ (d.h. gleich ;) oder eine geklammerte Folge von Statements;
- oder eine Zuweisung, eine Lese- oder Schreiboperation;
- eine (einseitige oder zweiseitige) bedingte Verzweigung;
- oder eine Schleife.

Ausdrücke

```
expr    ::=   number | name | ( expr ) |
              unop expr | expr binop expr
unop    ::=   -
binop   ::=   - | + | * | / | %
```

- Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- oder ein unärer Operator, angewandt auf einen Ausdruck,
- oder ein binärer Operator, angewand auf zwei Argumentausdrücke.
- Einziger unärer Operator ist (bisher) die Negation.
- Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganzzahlige) Division und Modulo.

Ausdrücke

expr	::=	number | name | (expr) |
		unop expr | expr binop expr
unop	::=	-
binop	::=	- | + | * | / | %

- Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- oder ein unärer Operator, angewandt auf einen Ausdruck,
- oder ein binärer Operator, angewand auf zwei Argumentausdrücke.
- Einziger unärer Operator ist (bisher) die Negation.
- Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganzzahlige) Division und Modulo.

Bedingungen

cond	::=	true | false | (cond) |
		expr comp expr |
		bunop cond | cond bbinop cond
comp	::=	== | != | <= | < | >= | >
bunop	::=	!
bbinop	::=	&& | ||

- Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein Wahrheitswert – vom Typ `boolean`).
- Bedingungen sind darum Konstanten, Vergleiche
- oder logische Verknüpfungen anderer Bedingungen.

Bedingungen

cond	::=	true | false | (cond) |
		expr comp expr |
		bunop cond | cond bbinop cond
comp	::=	== | != | <= | < | >= | >
bunop	::=	!
bbinop	::=	&& | ||

- Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein Wahrheitswert – vom Typ `boolean`).
- Bedingungen sind darum Konstanten, Vergleiche
- oder logische Verknüpfungen anderer Bedingungen.

Beispiel

```
int x;
x = read();
if (x > 0)
    write(1);
else
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume.

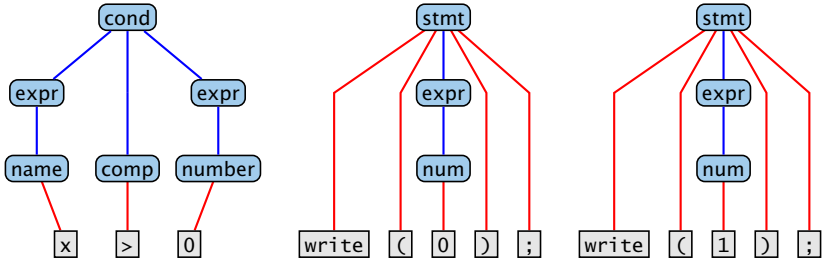
Beispiel

```
int x;
x = read();
if (x > 0)
    write(1);
else
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume.

Syntaxbäume

Syntaxbäume für `x > 0` sowie `write(0);` und `write(1);`

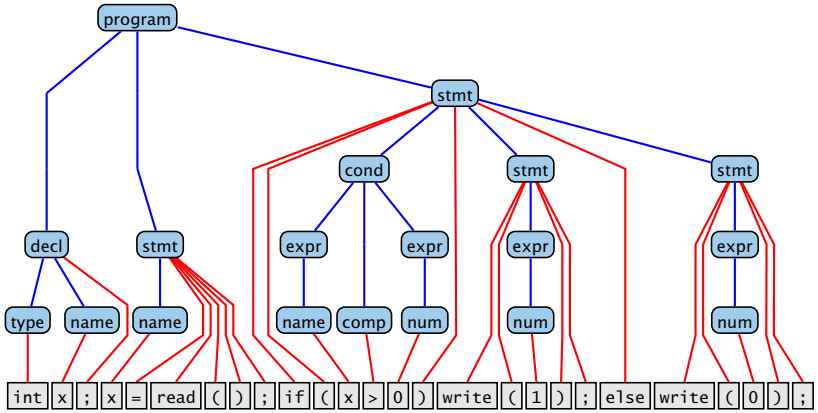


Blätter: Wörter/Tokens

innere Knoten: Namen von Programmbestandteilen

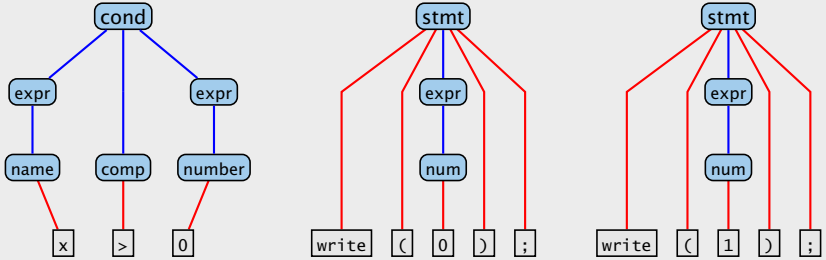
Beispiel

Der komplette Syntaxbaum unseres Beispiels:



Syntaxbäume

Syntaxbäume für `x > 0` sowie `write(0);` und `write(1);`

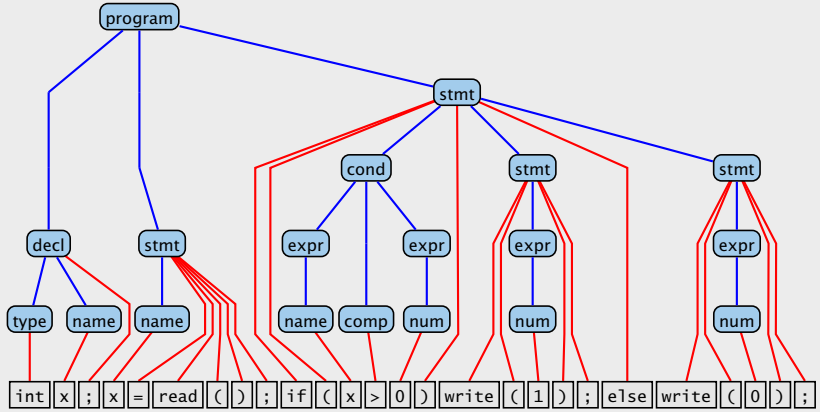


Blätter: Wörter/Tokens

innere Knoten: Namen von Programmbestandteilen

Beispiel

Der komplette Syntaxbaum unseres Beispiels:



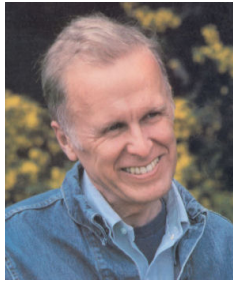
Bemerkungen

- Die vorgestellte Methode der Beschreibung von Syntax heißt EBNF-Notation (Extended Backus Naur Form Notation).
- Ein anderer Name dafür ist erweiterte kontextfreie Grammatik (↑Linguistik, ↑Automatentheorie).
- Linke Seiten von Regeln heißen auch Nichtterminale.
- Tokens heißen auch Terminale.

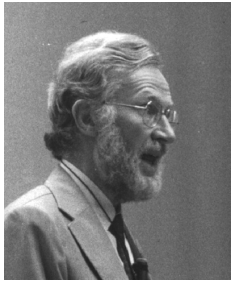
Bemerkungen

- Die vorgestellte Methode der Beschreibung von Syntax heißt EBNF-Notation (Extended Backus Naur Form Notation).
- Ein anderer Name dafür ist erweiterte kontextfreie Grammatik (↑Linguistik, ↑Automatentheorie).
- Linke Seiten von Regeln heißen auch Nichtterminale.
- Tokens heißen auch Terminale.

Noam Chomsky,
MIT



John Backus, IBM
Turing Award
(Erfinder von Fortran)



Peter Naur,
Turing Award
(Erfinder von Algol60)

Kontextfreie Grammatiken

Achtung:

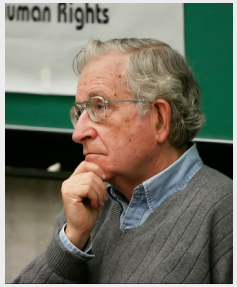
- Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nichtterminale enthalten.
- Deshalb sind kontextfreie Grammatiken mächtiger als reguläre Ausdrücke.

Beispiel:

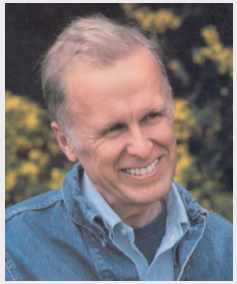
$$\mathcal{L} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

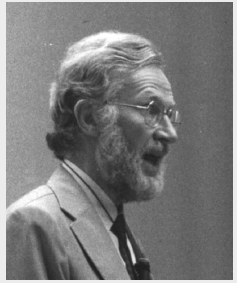
$$A \quad ::= \quad (\; a \; A \; b \;)?$$



Noam Chomsky,
MIT



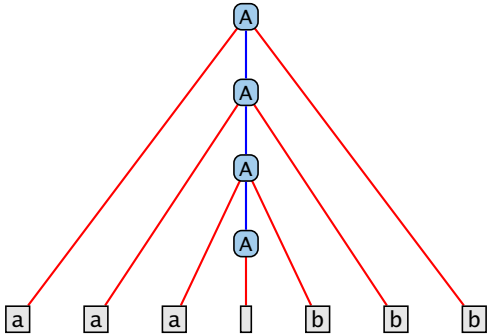
John Backus, IBM
Turing Award
(Erfinder von Fortran)



Peter Naur,
Turing Award
(Erfinder von Algol60)

Kontextfreie Grammatiken

Syntaxbaum für das Wort aaabbb:



Für $\mathcal{L}$ gibt es aber keinen regulären Ausdruck (↑Automatentheorie).

Kontextfreie Grammatiken

Achtung:

- Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nichtterminale enthalten.
- Deshalb sind kontextfreie Grammatiken mächtiger als reguläre Ausdrücke.

Beispiel:

$$\mathcal{L} = \{\epsilon, \text{ab}, \text{aabb}, \text{aaabbb}, \ldots\}$$

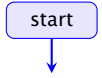
lässt sich mithilfe einer Grammatik beschreiben:

$$\text{A} \quad ::= \quad (\ \text{a A b}\ )?$$

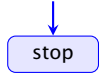
4 Kontrollflussdiagramme

In welcher Weise, Programmteile nacheinander ausgeführt werden kann anschaulich durch Kontrollflussdiagramme dargestellt werden.

Zutaten:

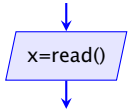


Startknoten

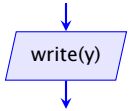


Endknoten

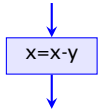
4 Kontrollflussdiagramme



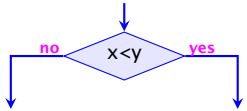
Eingabe



Ausgabe

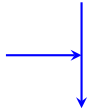


Zuweisung



bedingte Verzweigung

Kante



Zusammenlauf

4 Kontrollflussdiagramme

In welcher Weise, Programmteile nacheinander ausgeführt werden kann anschaulich durch Kontrollflussdiagramme dargestellt werden.

Zutaten:

Startknoten

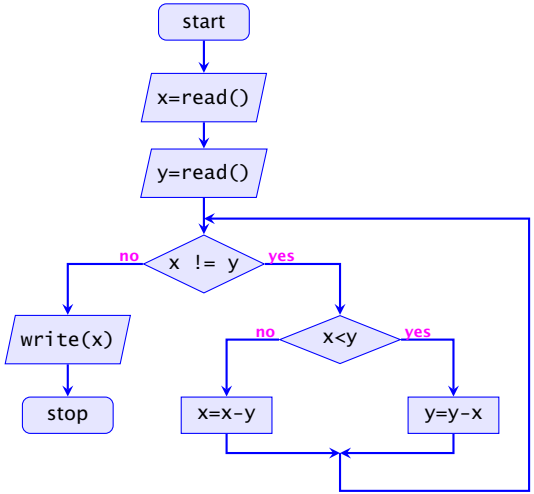
Endknoten

4 Kontrollflussdiagramme

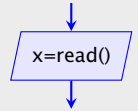
Beispiel:

```
int x, y;
x = read();
y = read();
while (x != y) {
    if (x < y)
        y = y - x;
    else
        x = x - y;
}
write(x);
```

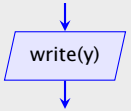
GGT



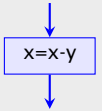
4 Kontrollflussdiagramme



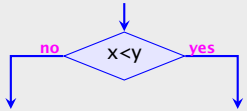
Eingabe



Ausgabe

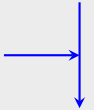


Zuweisung



bedingte Verzweigung

Kante



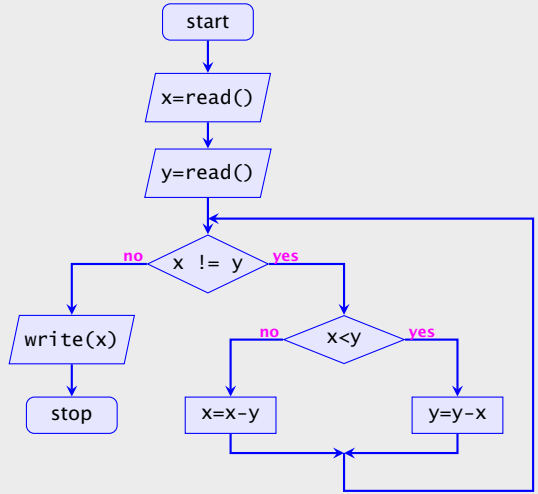
Zusammenlauf

4 Kontrollflussdiagramme

Beispiel:

```
int x, y;
x = read();
y = read();
while (x != y) {
    if (x < y)
        y = y - x;
    else
        x = x - y;
}
write(x);
```

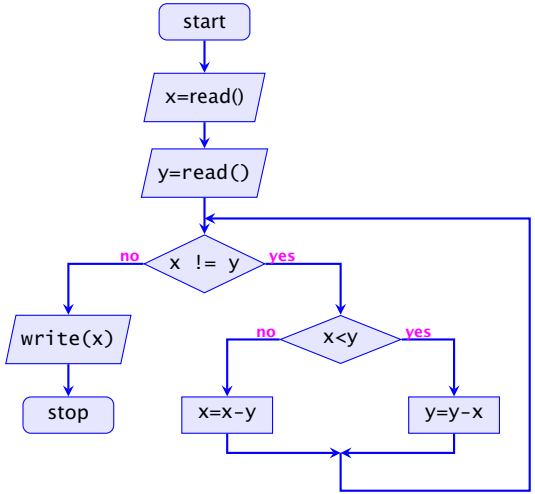
GGT



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

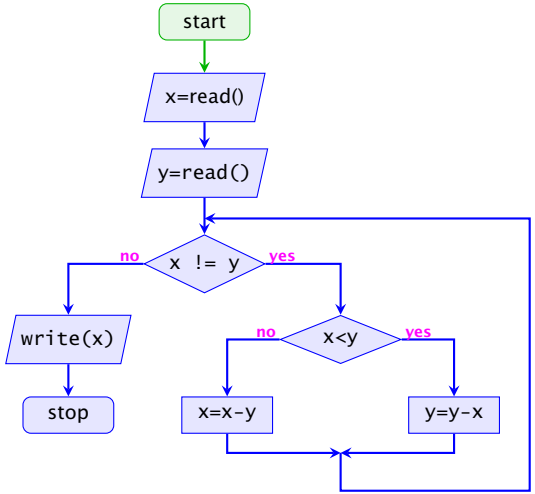
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

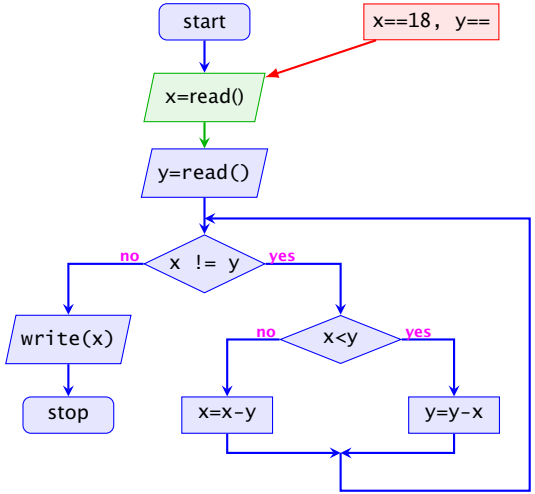
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

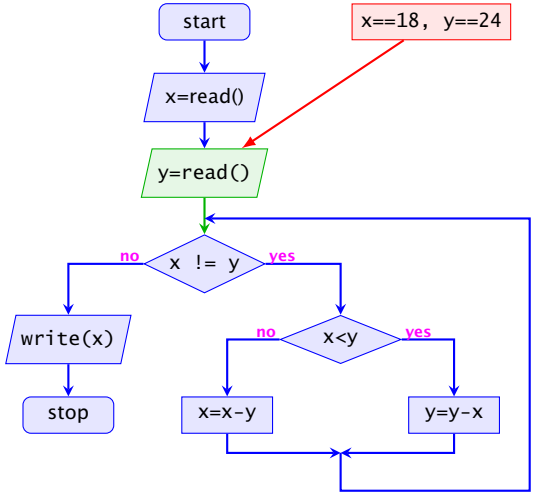
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

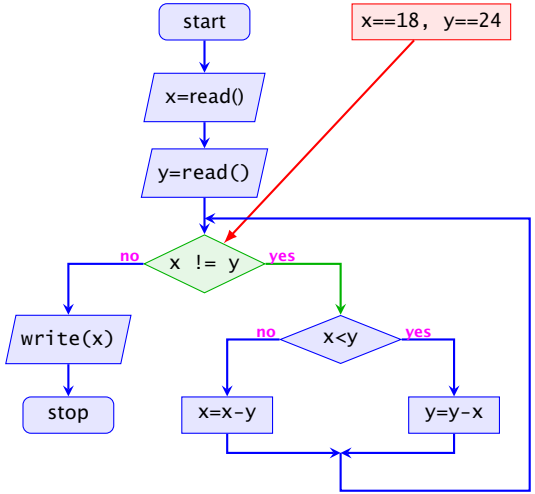
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

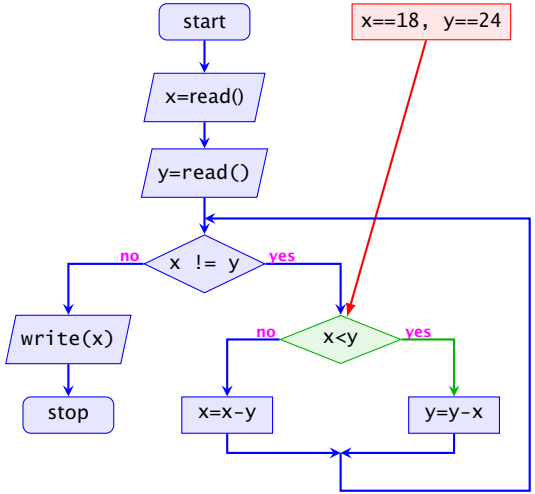
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

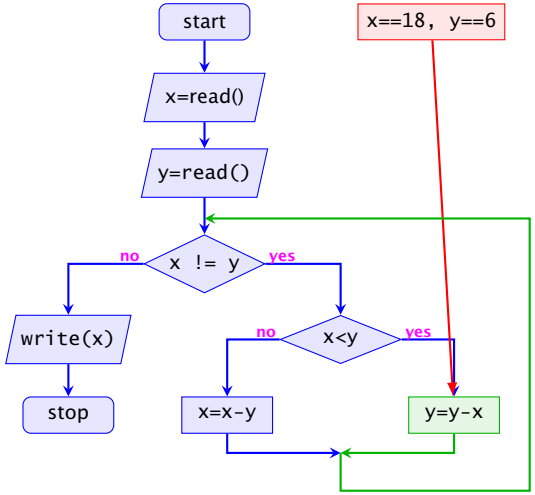
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

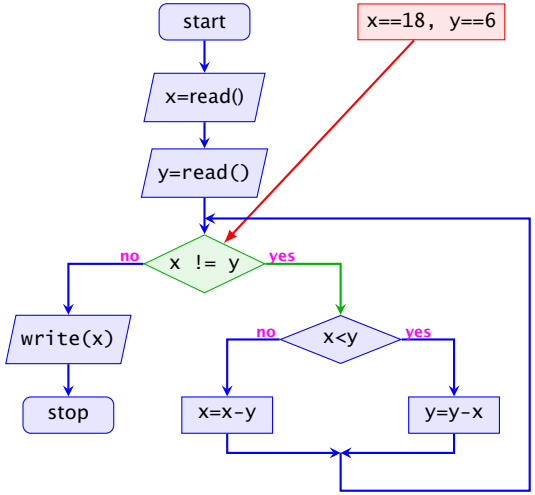
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

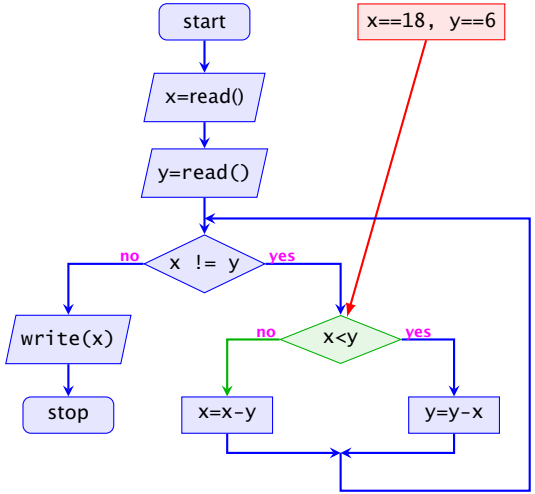
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

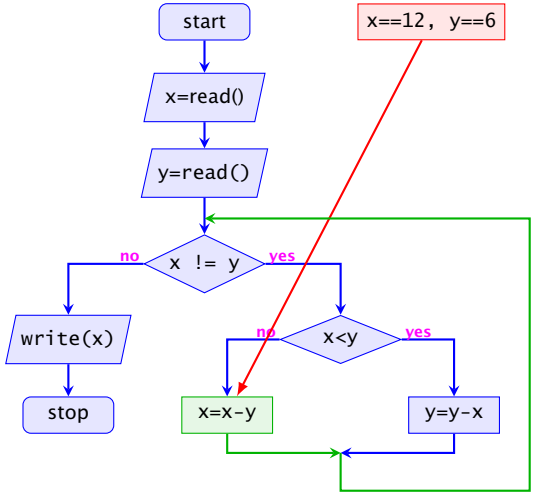
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

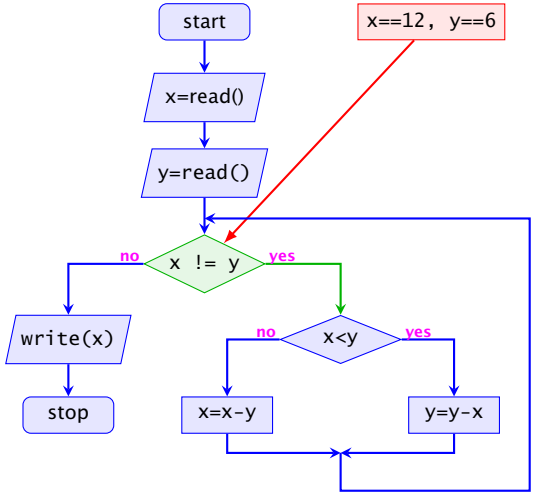
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

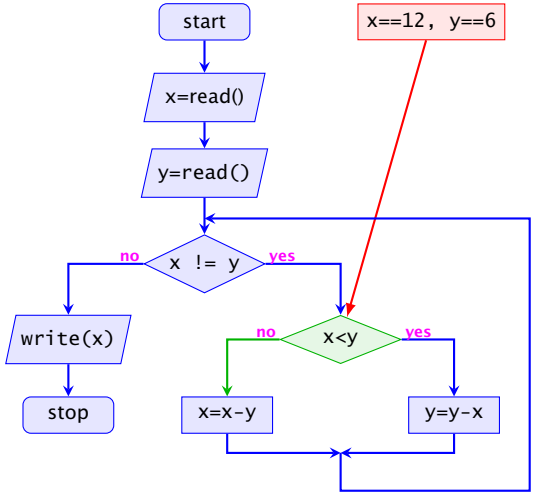
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

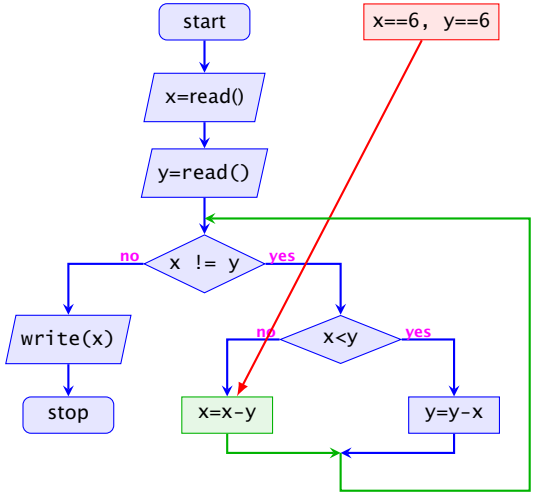
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

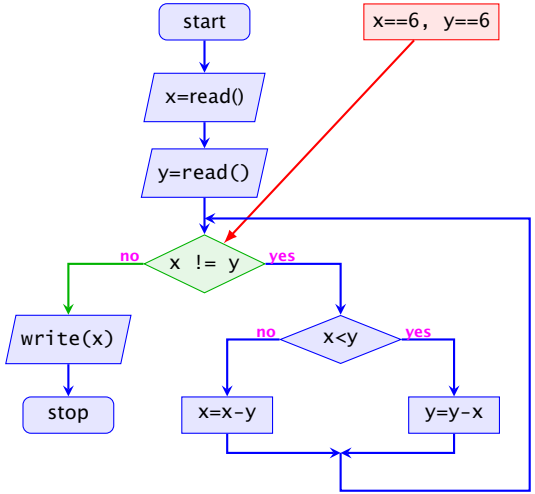
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

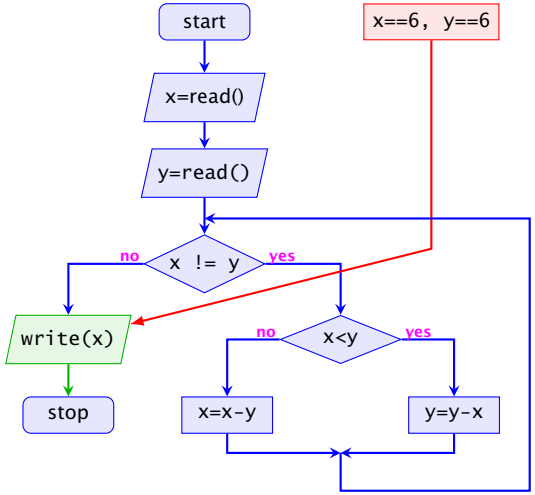
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

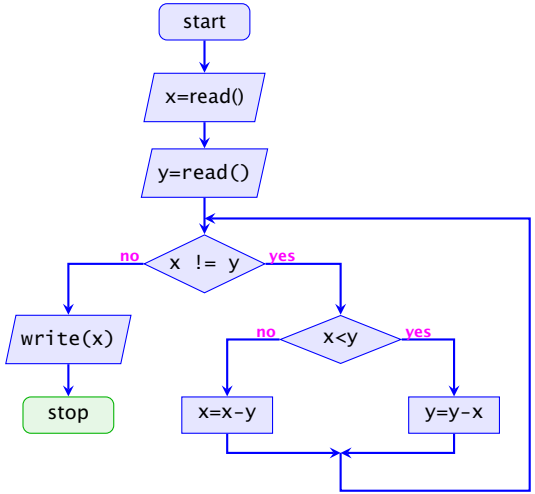
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

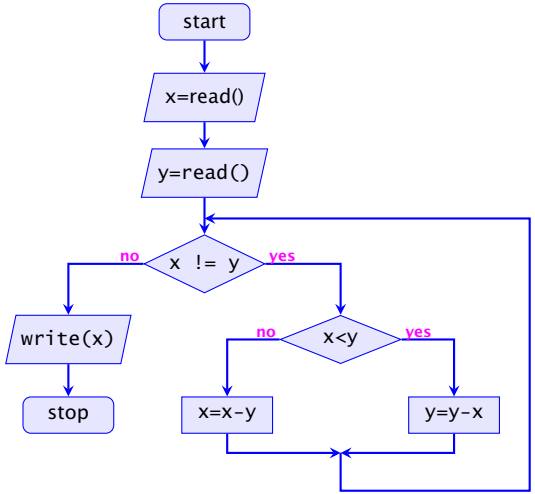
4 Kontrollflussdiagramme



4 Kontrollflussdiagramme

- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

4 Kontrollflussdiagramme



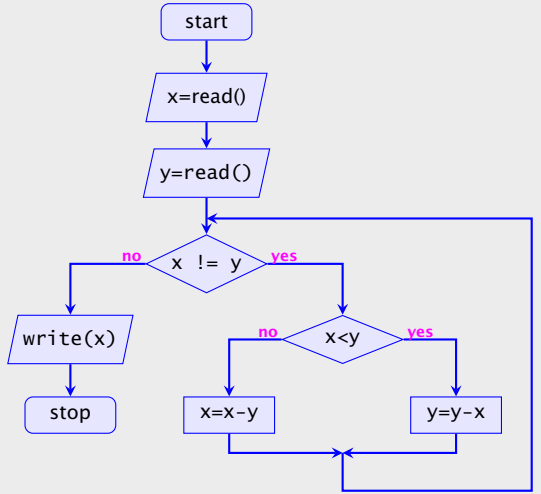
4 Kontrollflussdiagramme

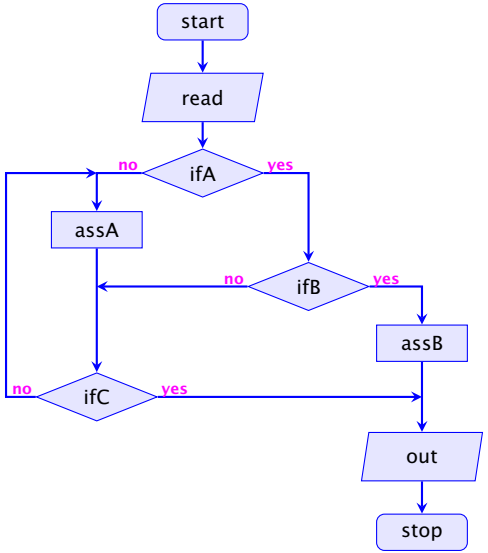
- Die Ausführung des Programms entspricht einem Pfad durch das Kontrollflussdiagramm vom Startknoten zum Endknoten.
- Die Deklaration von Variablen muss man sich am Startknoten vorstellen.
- Die auf dem Pfad liegenden Knoten (außer Start- und Endknoten) sind Operationen bzw. auszuwertende Bedingungen.
- Um den Nachfolger an einem Verzweigungsknoten zu bestimmen, muss die Bedingung mit den aktuellen Werten der Variablen ausgewertet werden. (↑operationelle Semantik)

4 Kontrollflussdiagramme

- zu jedem MiniJava-Programm lässt sich ein Kontrollflussdiagramm konstruieren;
- die Umkehrung gilt auch, liegt aber nicht sofort auf der Hand

4 Kontrollflussdiagramme





4 Kontrollflussdiagramme

- zu jedem MiniJava-Programm lässt sich ein Kontrollflussdiagramm konstruieren;
- die Umkehrung gilt auch, liegt aber nicht sofort auf der Hand

5 Mehr Java

Java ist statisch typisiert, d.h., Variablen, Ergebnisse von Ausdrücken, etc. haben ein Datentyp, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- Basistypen / Primitive Datentypen
  `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
- Referenzdatentypen
  kann man auch selber definieren

Java ist statisch typisiert, d.h., Variablen, Ergebnisse von Ausdrücken, etc. haben ein Datentyp, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- Basistypen / Primitive Datentypen
  `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `boolean`
- Referenzdatentypen
  kann man auch selber definieren

Beispiel – Statische Typisierung

```
a = 5
a = a + 1
a = "Hello World." # a is now a string
a = a + 1          # runtime error
```

Python

```
int a;
a = 5;
a = "Hello World." // will not compile
```

Java

5 Mehr Java

Java ist statisch typisiert, d.h., Variablen, Ergebnisse von Ausdrücken, etc. haben ein Datentyp, der schon bei der Kompilierung festgelegt wird.

Java unterscheidet zwei Arten von Typen:

- Basistypen / Primitive Datentypen
  byte, char, short, int, long, float, double, boolean
- Referenzdatentypen
  kann man auch selber definieren

5.1 Basistypen

Primitive Datentypen

- Zu jedem Basistypen gibt es eine Menge möglicher Werte.
- Jeder Wert eines Basistyps benötigt den gleichen Platz, um ihn im Rechner zu repräsentieren.
- Der Platz wird in Bit gemessen.

Wie viele Werte kann man mit n Bit darstellen?

Primitive Datentypen – Ganze Zahlen

Es gibt vier Sorten ganzer Zahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*
byte	8	−128	127
short	16	−32 768	32 767
int	32	−2 147 483 648	2 147 483 647
long	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie byte oder short spart Platz.

5.1 Basistypen

Primitive Datentypen

- Zu jedem Basistypen gibt es eine Menge möglicher Werte.
- Jeder Wert eines Basistyps benötigt den gleichen Platz, um ihn im Rechner zu repräsentieren.
- Der Platz wird in Bit gemessen.

Wie viele Werte kann man mit n Bit darstellen?

Primitive Datentypen – Ganze Zahlen

Literale:

- dezimale Notation
- hexadezimale Notation (Präfix 0x oder 0X)
- oktale Notation (Präfix 0)
- binäre Notation (Präfix 0b oder 0B)
- Suffix l☹ oder L für long
- '_' um Ziffern zu gruppieren

Beispiele

- 192, 0b11000000, 0xC0, 0300 sind alle gleich
- 20_000L, 0xABFF_0078L
- 09, 0x_FF sind ungültig

Primitive Datentypen – Ganze Zahlen

Es gibt vier Sorten ganzer Zahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*
byte	8	−128	127
short	16	−32 768	32 767
int	32	−2 147 483 648	2 147 483 647
long	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie byte oder short spart Platz.

Primitive Datentypen – Ganze Zahlen

Literale:

- dezimale Notation
- hexadezimale Notation (Präfix 0x oder 0X)
- oktale Notation (Präfix 0)
- binäre Notation (Präfix 0b oder 0B)
- Suffix l☹ oder L für long
- '_' um Ziffern zu gruppieren

Beispiele

- 192, 0b11000000, 0xC0, 0300 sind alle gleich
- 20_000L, 0xABFF_0078L
- 09, 0x_FF sind ungültig

Primitive Datentypen – Ganze Zahlen

Es gibt vier Sorten ganzer Zahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*
byte	8	−128	127
short	16	−32 768	32 767
int	32	−2 147 483 648	2 147 483 647
long	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Die Benutzung kleinerer Typen wie byte oder short spart Platz.

Primitive Datentypen – Ganze Zahlen

Achtung: Java warnt nicht vor Überlauf/Unterlauf!!!

Beispiel:

```
1 int x = 2147483647; // groesstes int
2 x = x + 1;
3 write(x);
```

liefert: -2147483648

Primitive Datentypen – Ganze Zahlen

Literale:

- dezimale Notation
- hexadezimale Notation (Präfix 0x oder 0X)
- oktale Notation (Präfix 0)
- binäre Notation (Präfix 0b oder 0B)
- Suffix l☹ oder L für long
- '_' um Ziffern zu gruppieren

Beispiele

- 192, 0b11000000, 0xC0, 0300 sind alle gleich
- 20_000L, 0xABFF_0078L
- 09, 0x_FF sind ungültig

Primitive Datentypen – Gleitkommazahlen

Es gibt zwei Sorten von Gleitkommazahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*	*signifikante Stellen*
`float`	32	ca. $-3.4 \cdot 10^{38}$	ca. $3.4 \cdot 10^{38}$	ca. 7
`double`	64	ca. $-1.7 \cdot 10^{308}$	ca. $1.7 \cdot 10^{308}$	ca. 15

$$x = s \cdot m \cdot 2^e \qquad \text{mit } 1 \leq m < 2$$

- Vorzeichen s: 1 bit
- reduzierte Mantisse $m - 1$: 23 bit (`float`), 52 bit (`double`)
- Exponent e: 8 bit (`float`), 11 bit (`double`)

Primitive Datentypen – Ganze Zahlen

Achtung: Java warnt nicht vor Überlauf/Unterlauf!!!

Beispiel:

```
1 int x = 2147483647; // groesstes int
2 x = x + 1;
3 write(x);
```

liefert: `-2147483648`

Primitive Datentypen – Gleitkommazahlen

Literale:

- dezimale Notation.
- dezimale Exponentialschreibweise (e, E für Exponent)
- hexadeximale Exponentialschreibweise. (Präfix 0x oder 0X, p oder P für Exponent)
- Suffix f oder F für `float`, Suffix d oder D für `double` (default is `double`)

Beispiele

- `640.5F` == `0x50.1p3f`
- `3.1415` == `314.15E-2`
- `0x1e3_dp0`, `1e3d`
- `0x1e3d`, `1e3_d`

Primitive Datentypen – Gleitkommazahlen

Es gibt zwei Sorten von Gleitkommazahlen:

Typ	*Platz*	*kleinster Wert*	*größter Wert*	*signifikante Stellen*
`float`	32	ca. $-3.4 \cdot 10^{38}$	ca. $3.4 \cdot 10^{38}$	ca. 7
`double`	64	ca. $-1.7 \cdot 10^{308}$	ca. $1.7 \cdot 10^{308}$	ca. 15

$$x = s \cdot m \cdot 2^e \qquad \text{mit } 1 \leq m < 2$$

- Vorzeichen s: 1 bit
- reduzierte Mantisse $m - 1$: 23 bit (`float`), 52 bit (`double`)
- Exponent e: 8 bit (`float`), 11 bit (`double`)

Primitive Datentypen – Gleitkommazahlen

- Überlauf/Unterlauf bei Berechnungen liefert `Infinity`, bzw. `-Infinity`
- Division Null durch Null, Wurzel aus einer negativen Zahl etc. liefert `NaN`

Primitive Datentypen – Gleitkommazahlen

Literale:

- dezimale Notation.
- dezimale Exponentialschreibweise (`e`, `E` für Exponent)
- hexadeximale Exponentialschreibweise. (Präfix `0x` oder `0X`, `p` oder `P` für Exponent)
- Suffix `f` oder `F` für `float`, Suffix `d` oder `D` für `double` (default is `double`)

Beispiele

- `640.5F` == `0x50.1p3f`
- `3.1415` == `314.15E-2`
- `0x1e3_dp0`, `1e3d`
- `0x1e3d`, `1e3_d`

Weitere Basistypen

Typ	*Platz*	*Werte*
`boolean`	1	`true,false`
`char`	16	all Unicode-Zeichen

Unicode ist ein Zeichensatz, der alle irgendwo auf der Welt gängigen Alphabete umfasst, also zum Beispiel:

- die Zeichen unserer Tastatur (inklusive Umlaute);
- die chinesischen Schriftzeichen;
- die ägyptischen Hieroglyphen ...

Literale:

- `char`-Literale schreibt man in Hochkommas: `'A'`, `'\u00ED'`,`';'`, `'\n'`.
- `boolean`-Literale sind `true` und `false`.

Primitive Datentypen – Gleitkommazahlen

- Überlauf/Unterlauf bei Berechnungen liefert `Infinity`, bzw. `-Infinity`
- Division Null durch Null, Wurzel aus einer negativen Zahl etc. liefert `NaN`

5.2 Strings

Der Datentyp `String` für Wörter ist ein Referenzdatentyp
(genauer eine Klasse (dazu kommen wir später)).

Hier nur drei Eigenschaften:

- Literale vom Typ `String` haben die Form `"Hello World!"`;
- Man kann Wörter in Variablen vom Typ `String` abspeichern;
- Man kann Wörter mithilfe des Operators `'+'` konkatenieren.

Beispiel

```
String s0 = "";
String s1 = "Hel";
String s2 = "lo Wo";
String s3 = "rld!";

write(s0 + s1 + s2 + s3);
```

...liefert: `Hello World!`

5.2 Strings

Der Datentyp `String` für Wörter ist ein Referenzdatentyp (genauer eine Klasse (dazu kommen wir später)).

Hier nur drei Eigenschaften:

- Literale vom Typ `String` haben die Form `"Hello World!"`;
- Man kann Wörter in Variablen vom Typ `String` abspeichern;
- Man kann Wörter mithilfe des Operators `'+'` konkatenieren.

5.3 Auswertung von Ausdrücken

Funktionen in Java bekommen Parameter/Argumente als Input, und liefern als Output den Wert eines vorbestimmten Typs. Zum Beispiel könnte man eine Funktion

```
int min(int a, int b)
```

implementieren, die das Minimum ihrer Argumente zurückliefert.

Operatoren sind spezielle vordefinierte Funktionen, die in Infix-Notation geschrieben werden (wenn sie binär sind):

```
a + b = +(a,b)
```


Beispiel

```
String s0 = "";
String s1 = "Hel";
String s2 = "lo Wo";
String s3 = "rld!";

write(s0 + s1 + s2 + s3);
```

...liefert: Hello World!

5.3 Auswertung von Ausdrücken

Ein Ausdruck ist eine Kombination von Literalen, Operatoren, Funktionen, Variablen und Klammern, die verwendet wird, um einen Wert zu berechnen.

Beispiele: (x z.B. vom Typ `int`)

- `7 + 4`
- `3 / 5 + 3`
- `min(3,x) + 20`
- `x = 7`
- `x *= 2`

5.3 Auswertung von Ausdrücken

Funktionen in Java bekommen Parameter/Argumente als Input, und liefern als Output den Wert eines vorbestimmten Typs. Zum Beispiel könnte man eine Funktion

```
int min(int a, int b)
```

implementieren, die das Minimum ihrer Argumente zurückliefert.

Operatoren sind spezielle vordefinierte Funktionen, die in Infix-Notation geschrieben werden (wenn sie binär sind):

```
a + b = +(a,b)
```

Operatoren

Unäre Operatoren:

symbol	*name*	*types*	*L/R*	*level*
++	Post-increment	(var) zahl, char	links	2
--	Post-decrement	(var) zahl, char	links	2
++	Pre-increment	(var) zahl, char	rechts	3
--	Pre-decrement	(var) zahl, char	rechts	3
+	unäres Plus	zahl, char	rechts	3
-	unäres Minus	zahl, char	rechts	3
!	Negation	boolean	rechts	3

5.3 Auswertung von Ausdrücken

Ein Ausdruck ist eine Kombination von Literalen, Operatoren, Funktionen, Variablen und Klammern, die verwendet wird, um einen Wert zu berechnen.

Beispiele: (`x` z.B. vom Typ `int`)

- `7 + 4`
- `3 / 5 + 3`
- `min(3,x) + 20`
- `x = 7`
- `x *= 2`

Achtung

Operatoren

Unäre Operatoren:

symbol	*name*	*types*	*L/R*	*level*
++	Post-increment	(var) zahl, char	links	2
--	Post-decrement	(var) zahl, char	links	2
++	Pre-increment	(var) zahl, char	rechts	3
--	Pre-decrement	(var) zahl, char	rechts	3
+	unäres Plus	zahl, char	rechts	3
-	unäres Minus	zahl, char	rechts	3
!	Negation	boolean	rechts	3

Prefix- und Postfixoperator

- Die Operatoranwendungen ++x und x++ inkrementieren beide den Wert der Variablen x (als Seiteneffekt).
- ++x tut das, bevor der Wert des Ausdrucks ermittelt wird (Pre-Increment).
- x++ tut das, nachdem der Wert ermittelt wurde (Post-Increment).
- `b = x++;` entspricht:

```
b = x;
x = x + 1;
```

- `b = ++x;` entspricht:

```
x = x + 1;
b = x;
```


Achtung

Prefix- und Postfixoperator

- Die Operatoranwendungen ++x und x++ inkrementieren beide den Wert der Variablen x (als Seiteneffekt).
- ++x tut das, bevor der Wert des Ausdrucks ermittelt wird (Pre-Increment).
- x++ tut das, nachdem der Wert ermittelt wurde (Post-Increment).
- `b = x++;` entspricht:

```
b = x;
x = x + 1;
```

- `b = ++x;` entspricht:

```
x = x + 1;
b = x;
```


Operatoren

Binäre arithmetische Operatoren:

byte, short, char werden nach int konvertiert

symbol	*name*	*types*	*L/R*	*level*
*	Multiplikation	zahl, char	links	4
/	Division	zahl, char	links	4
%	Modulo	zahl, char	links	4
+	Addition	zahl, char	links	5
-	Subtraktion	zahl, char	links	5

Konkatenation

symbol	*name*	*types*	*L/R*	*level*
+	Konkatenation	string	links	5

Operatoren

Vergleichsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
>	größer	zahl, char	links?	7
>=	größergleich	zahl, char	links?	7
<	kleiner	zahl, char	links?	7
<=	kleinergleich	zahl, char	links?	7
==	gleich	primitiv	links	8
!=	ungleich	primitiv	links	8

Operatoren

Binäre arithmetische Operatoren:

`byte`, `short`, `char` werden nach `int` konvertiert

symbol	*name*	*types*	*L/R*	*level*
*	Multiplikation	zahl, char	links	4
/	Division	zahl, char	links	4
%	Modulo	zahl, char	links	4
+	Addition	zahl, char	links	5
-	Subtraktion	zahl, char	links	5

Konkatenation

symbol	*name*	*types*	*L/R*	*level*
+	Konkatenation	string	links	5

Operatoren

Vergleichsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
>	größer	zahl, char	links?	7
>=	größergleich	zahl, char	links?	7
<	kleiner	zahl, char	links?	7
<=	kleinergleich	zahl, char	links?	7
==	gleich	primitiv	links	8
!=	ungleich	primitiv	links	8

Operatoren

Boolsche Operatoren:

symbol	*name*	*types*	*L/R*	*level*
&&	Und-Bedingung	boolean	links	12
||	Oder-Bedingung	boolean	links	13

Operatoren

Zuweisungsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
=	Zuweisung	var, wert	rechts	15
*=,/=, %=, +=, -=	Zuweisung	var, wert	rechts	15

Für die letzte Form gilt:

$$v \circ\!= a \iff v = (type(v))\ (v \circ a)$$

Operatoren

Boolsche Operatoren:

symbol	*name*	*types*	*L/R*	*level*
&&	Und-Bedingung	boolean	links	12
||	Oder-Bedingung	boolean	links	13

Operatoren

Zuweisungsoperatoren:

symbol	*name*	*types*	*L/R*	*level*
=	Zuweisung	var, wert	rechts	15
*=,/=, %=, +=, -=	Zuweisung	var, wert	rechts	15

Für die letzte Form gilt:

$$v \circeq a \iff v = (type(v))\ (v \circ a)$$

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen x erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen `x` erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen `x` erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen x erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen `x` erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen `x` erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

Operatoren

Warnung:

- Eine Zuweisung `x = y;` ist in Wahrheit ein Ausdruck.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variablen x erfolgt als Seiteneffekt.
- Das Semikolon `';'` hinter einem Ausdruck wirft nur den Wert weg.

Fatal für Fehler in Bedingungen:

```
boolean x = false;
if (x = true)
   write("Sorry! This must be an error ...");
```

5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- eine Variable (**var**),
- einen reinen Wert (**val**) oder
- void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- Typ, z.B.: **int**, und einen
- Wert, z.B.: **42**

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

5.3 Auswertung von Ausdrücken

Assoziativität

- Die Assoziativität entscheidet über die Reihenfolge bei Operatoren gleicher Priorität. (links = der linkeste Operator wird zuerst ausgeführt)
- Alle Operatoren einer Prioritätsgruppe haben dieselbe Assoziativität.
- Bis auf Zuweisungsoperatoren (=, +=, etc.) sind alle binären Operatoren linksassoziativ.
- unäre Operatoren, die ihr Argument rechts erwarten sind rechtsassoziativ
- unäre Operatoren, die ihr Argument links erwarten (postfix-Operatoren ++, --) sind linksassoziativ
- Der ternäre Bedingungsoperator (später) ist rechtsassoziativ

5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- eine Variable (**var**),
- einen reinen Wert (**val**) oder
- void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- Typ, z.B.: **int**, und einen
- Wert, z.B.: **42**

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

5.3 Auswertung von Ausdrücken

In Java werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- werte (rekursiv) alle Operanden von links nach rechts aus
- führe die Operation auf den Resultaten aus

Ausnahmen: ||, &&, und der ternäre Bedingungsoperator :?, werten nicht all Operanden aus (Kurzschlussauswertung).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

5.3 Auswertung von Ausdrücken

Die Auswertung eines Ausdrucks liefert

- eine Variable (**var**),
- einen reinen Wert (**val**) oder
- void (**void**)

In den ersten beiden Fällen hat der Ausdruck dann einen

- Typ, z.B.: **int**, und einen
- Wert, z.B.: **42**

Für z.B. Zuweisungen muss die Auswertung des Ausdrucks auf der linken Seite eine Variable ergeben!!!

5.3 Auswertung von Ausdrücken

In Java werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- werte (rekursiv) alle Operanden von links nach rechts aus
- führe die Operation auf den Resultaten aus

Ausnahmen: ||, &&, und der ternäre Bedingungsoperator :?, werten nicht all Operanden aus (Kurzschlussauswertung).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

5.3 Auswertung von Ausdrücken

In Java werden Unterausdrücke von links nach rechts ausgewertet. D.h. um den Wert einer Operation zu berechnen:

- werte (rekursiv) alle Operanden von links nach rechts aus
- führe die Operation auf den Resultaten aus

Ausnahmen: ||, &&, und der ternäre Bedingungsoperator :?, werten nicht all Operanden aus (Kurzschlussauswertung).

Man sollte nie Ausdrücke formulieren, deren Ergebnis von der Auswertungsreihenfolge abhängt!!!

5.3 Auswertung von Ausdrücken

Im folgenden betrachten wir Klammern als einen Operator der nichts tut:

symbol	*name*	*types*	*L/R*	*level*
( )	Klammerung	*	links	0

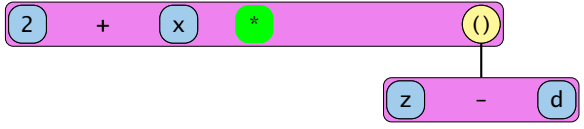
Beispiel: 2 + x * (z - d)

Beispiel: 2 + x * (z - d)

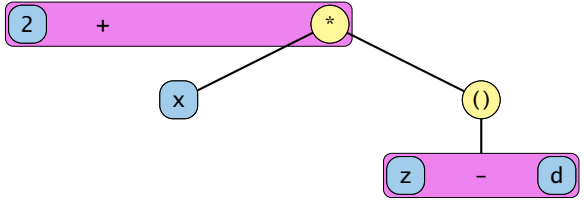
Beispiel: 2 + x * (z - d)

Beispiel: 2 + x * (z - d)

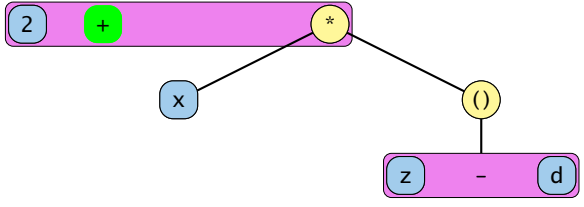
Beispiel: 2 + x * (z - d)



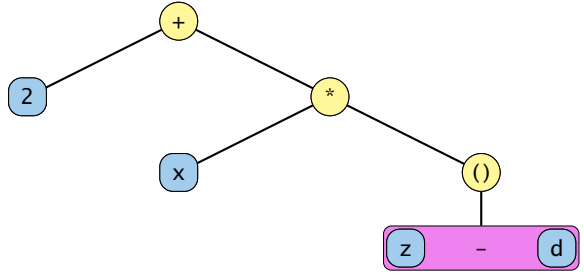
Beispiel: 2 + x * (z - d)



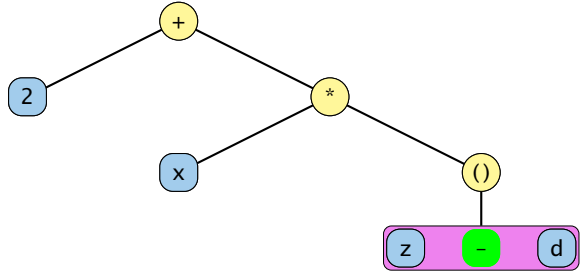
Beispiel: 2 + x * (z - d)



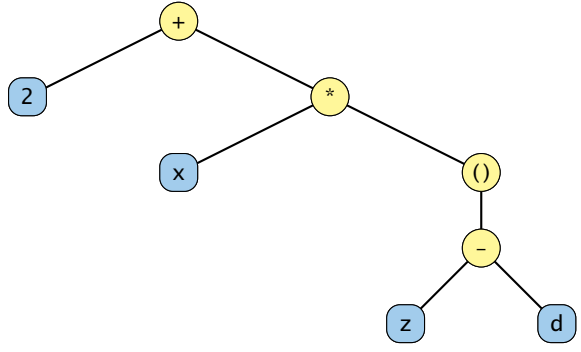
Beispiel: 2 + x * (z - d)



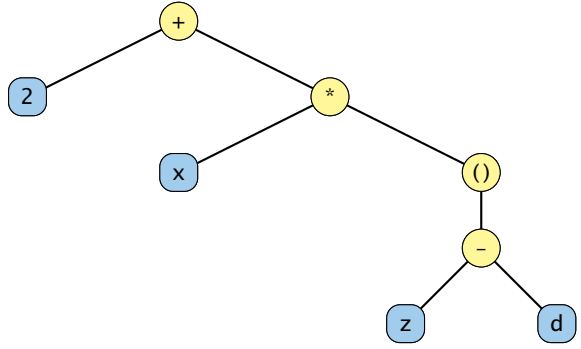
Beispiel: 2 + x * (z - d)



Beispiel: 2 + x * (z - d)

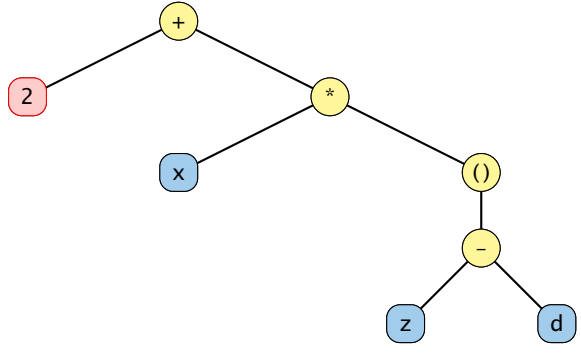


Beispiel: 2 + x * (z - d)



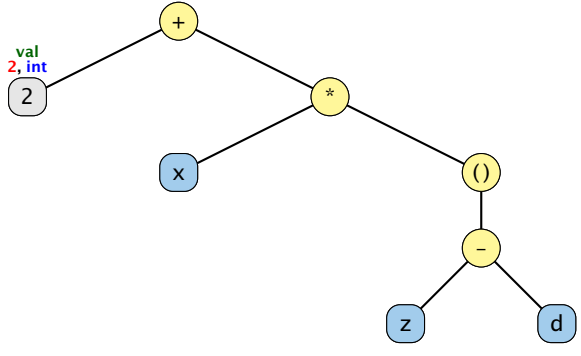
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



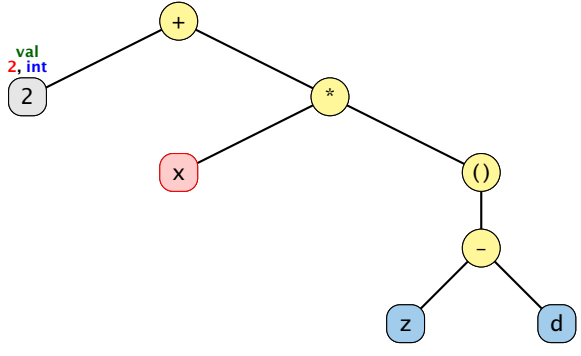
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



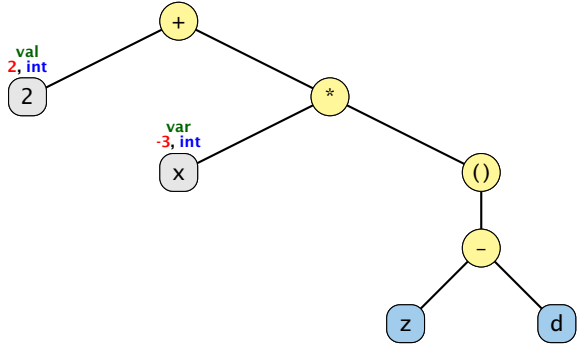
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



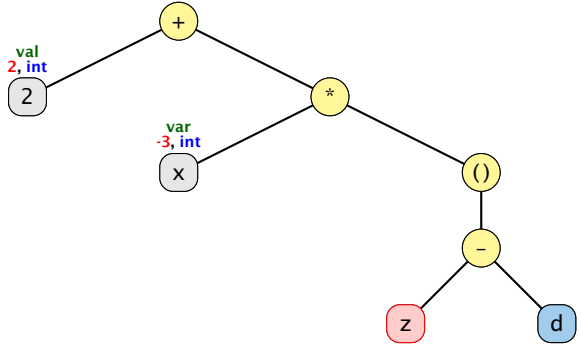
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



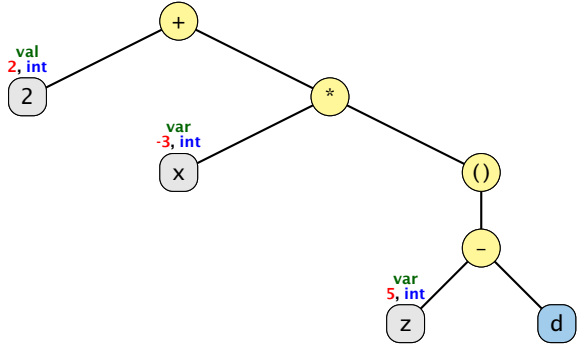
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



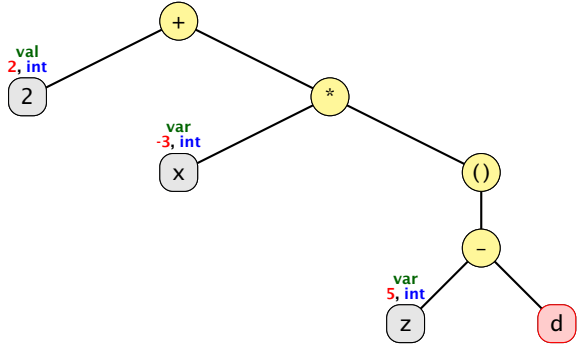
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



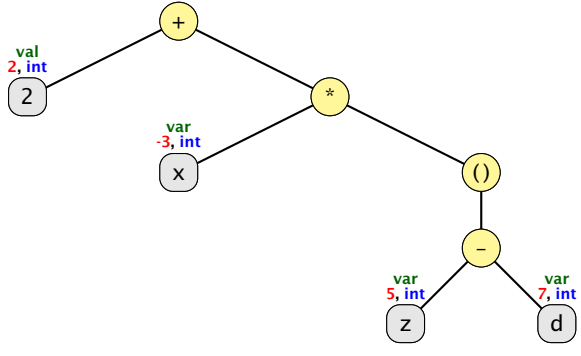
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



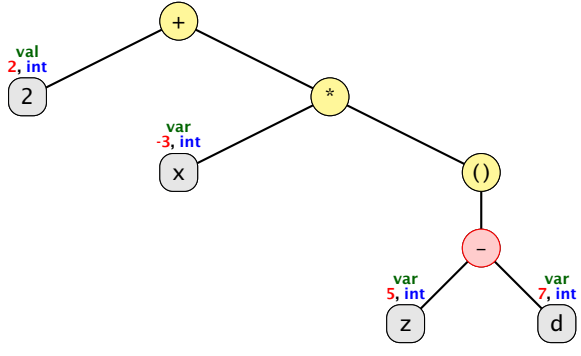
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



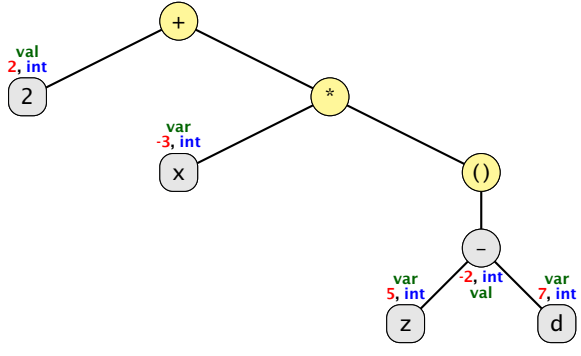
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



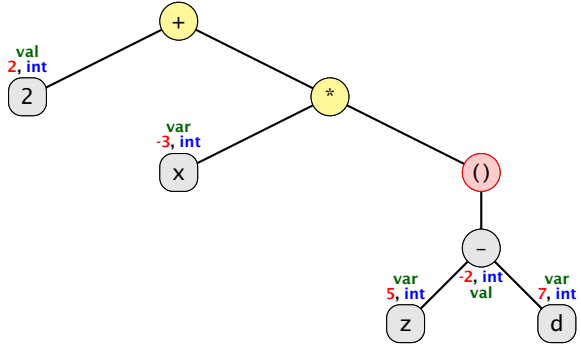
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



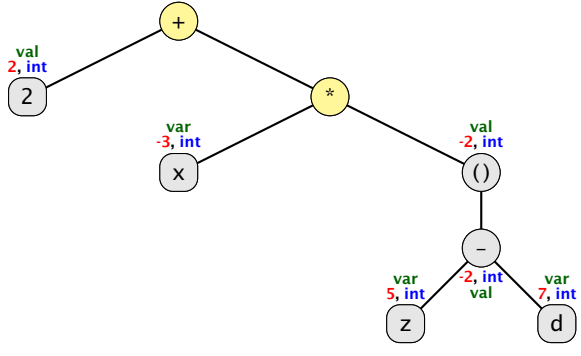
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



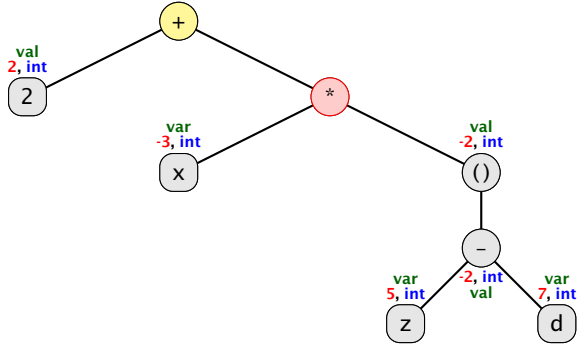
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



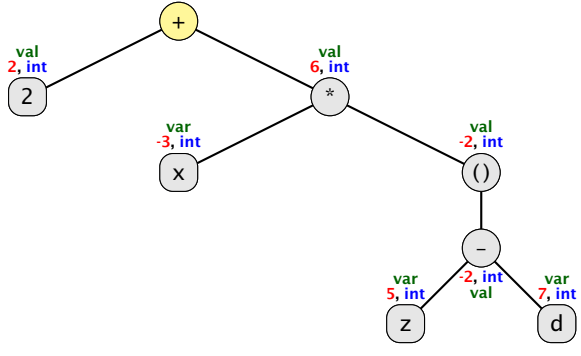
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



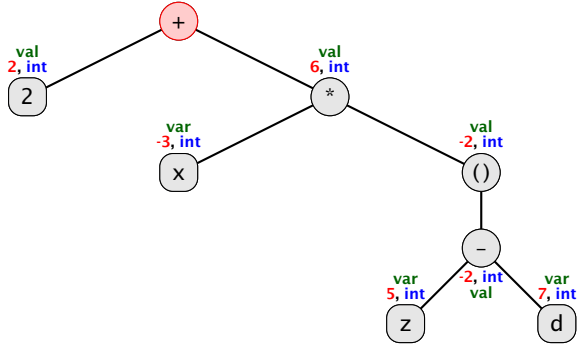
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



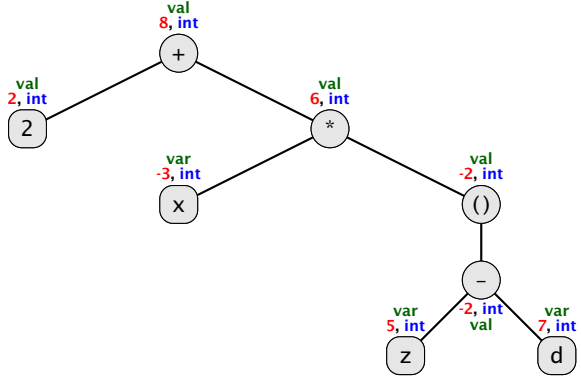
x -3 d 7 z 5

Beispiel: 2 + x * (z - d)



x -3 d 7 z 5

Beispiel: 2 + x * (z - d)

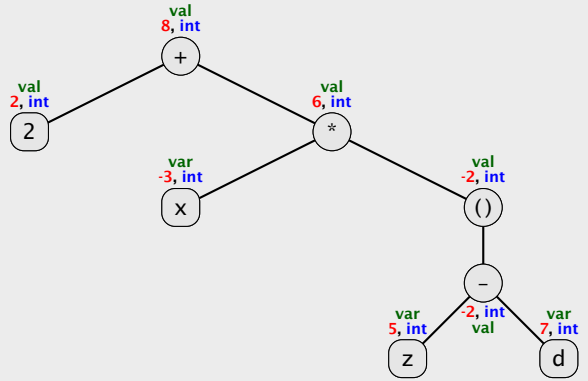


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

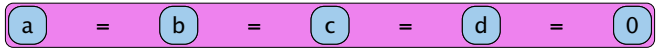
a	=	b	=	c	=	d	=	0

Beispiel: 2 + x * (z - d)

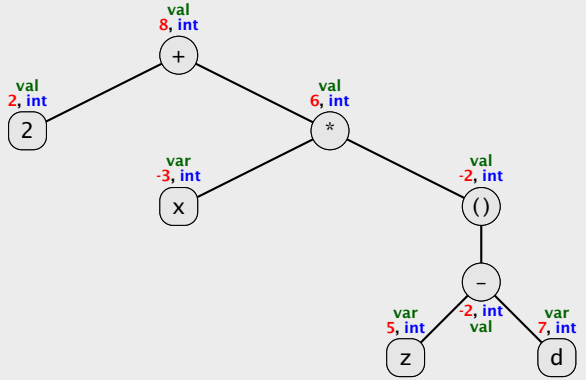


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

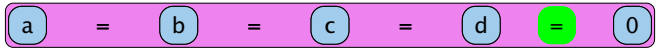


Beispiel: 2 + x * (z - d)

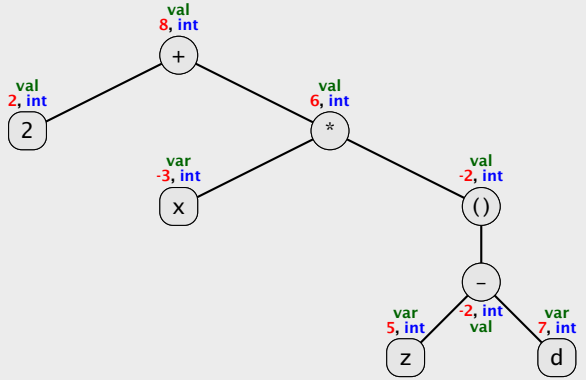


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

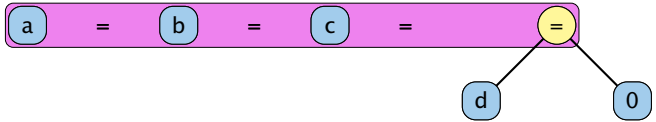


Beispiel: 2 + x * (z - d)

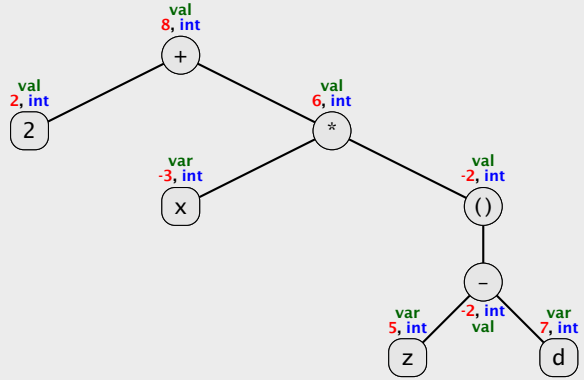


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

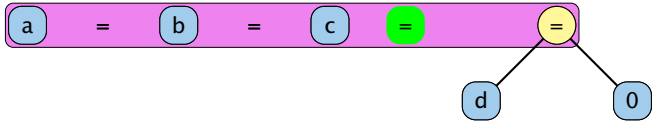


Beispiel: 2 + x * (z - d)

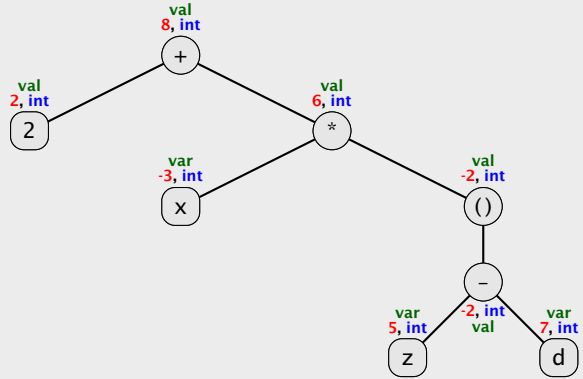


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

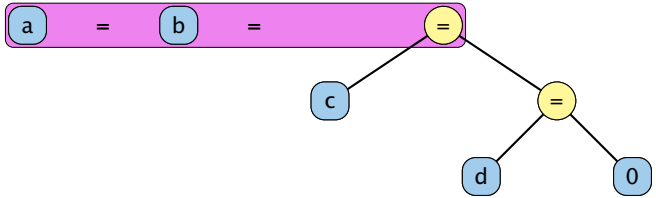


Beispiel: 2 + x * (z - d)

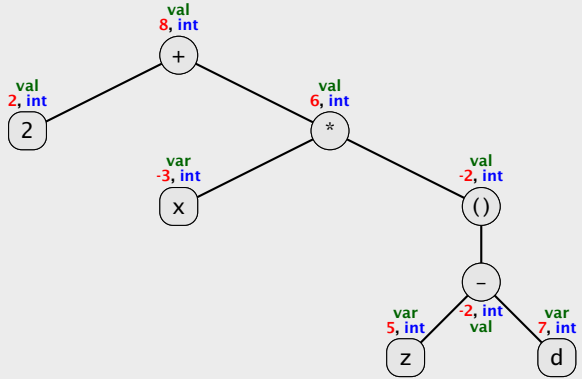


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

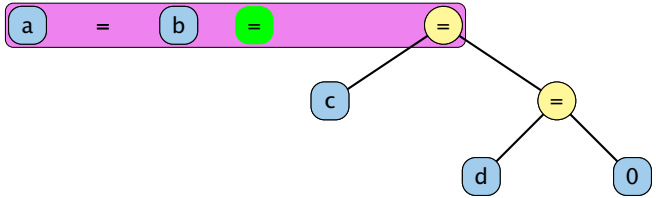


Beispiel: 2 + x * (z - d)

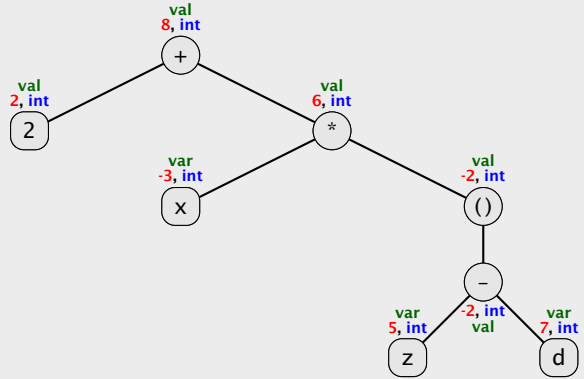


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

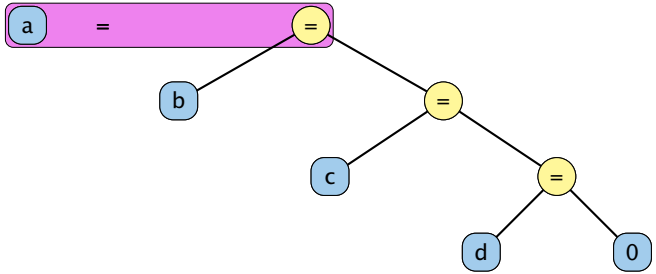


Beispiel: 2 + x * (z - d)

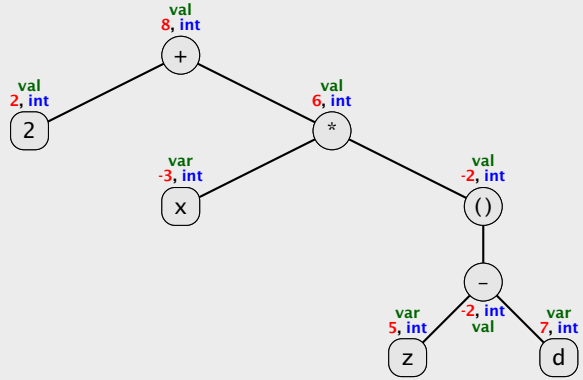


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

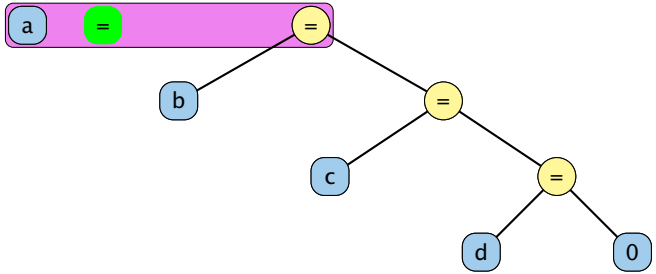


Beispiel: 2 + x * (z - d)

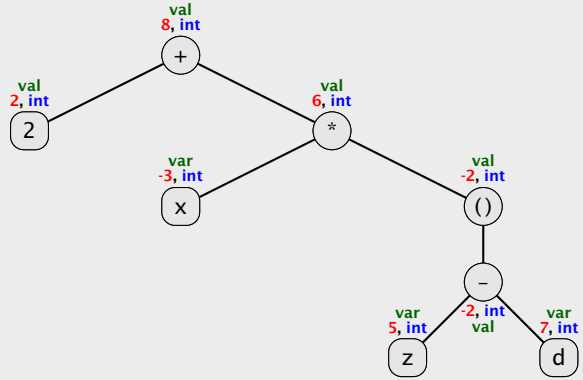


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

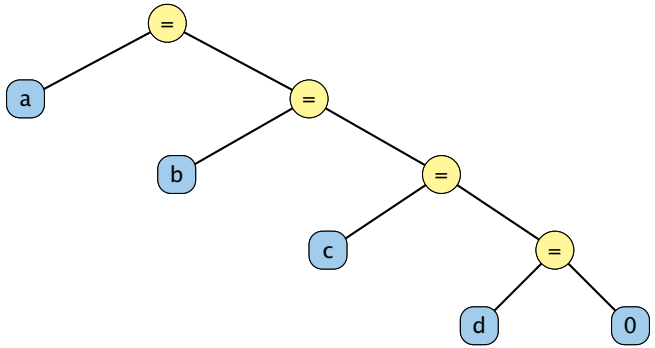


Beispiel: 2 + x * (z - d)

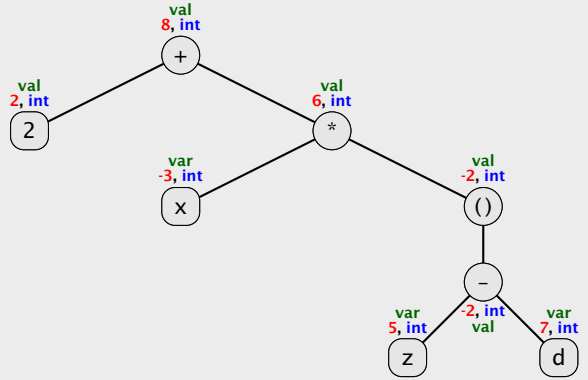


x -3 d 7 z 5

Beispiel: a = b = c = d = 0

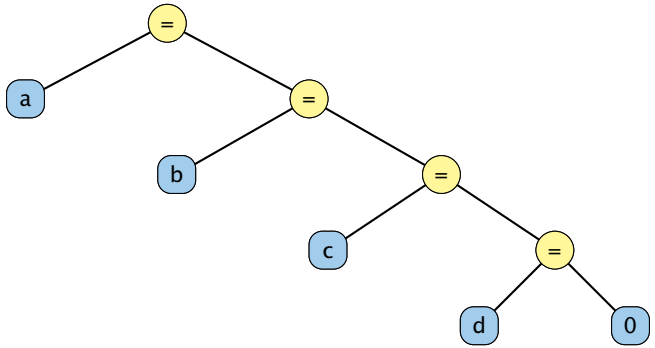


Beispiel: 2 + x * (z - d)



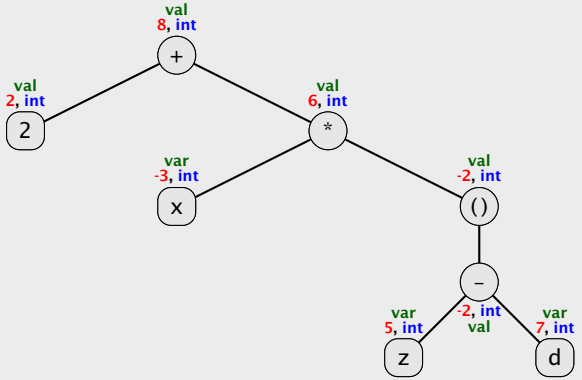
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



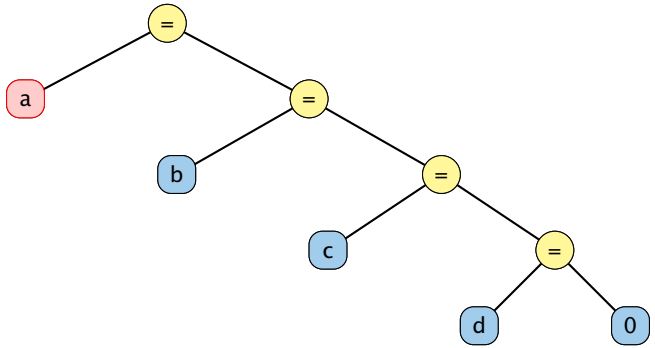
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



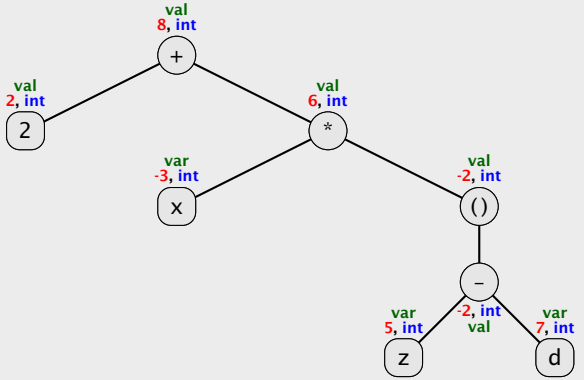
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



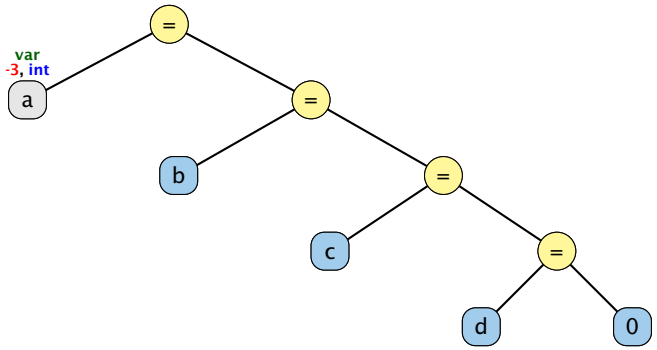
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



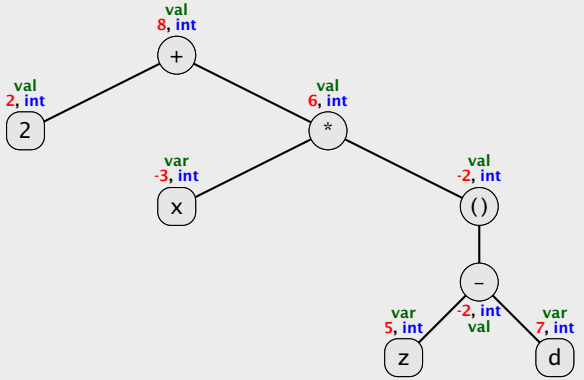
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



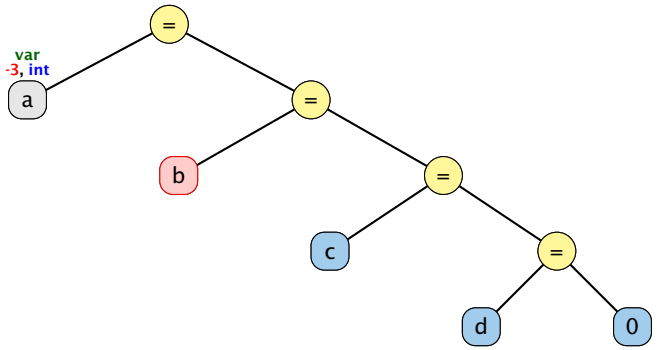
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



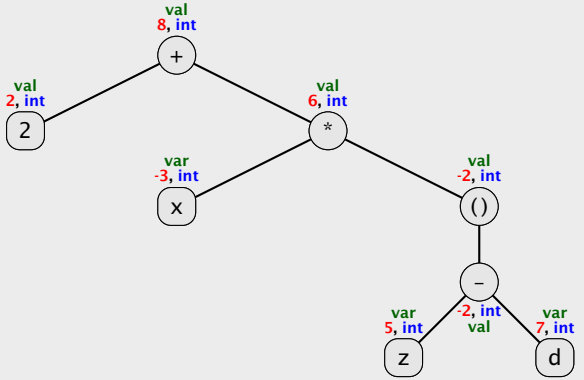
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



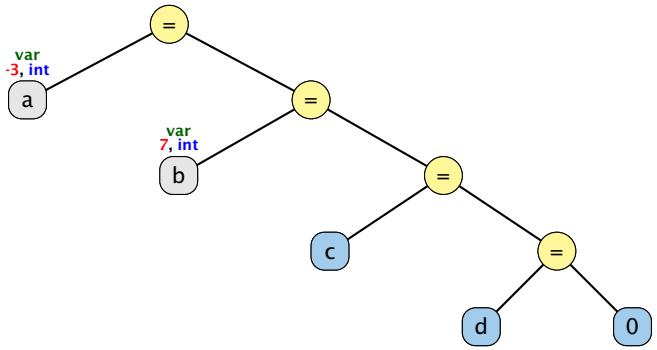
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



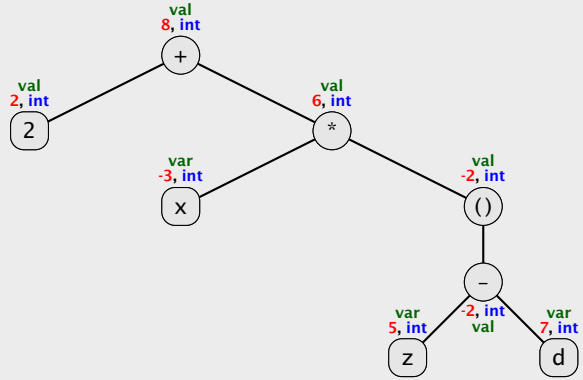
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



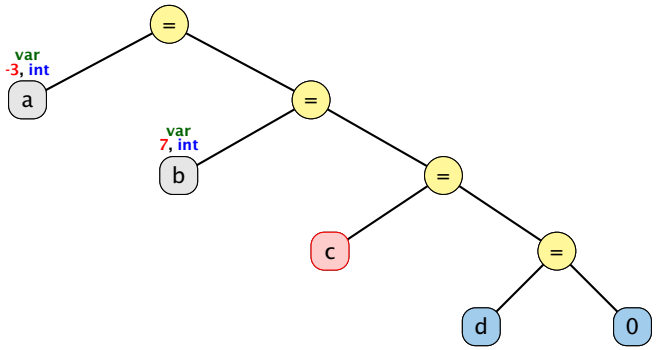
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



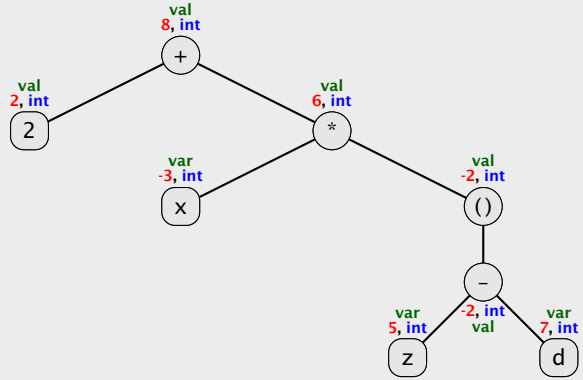
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



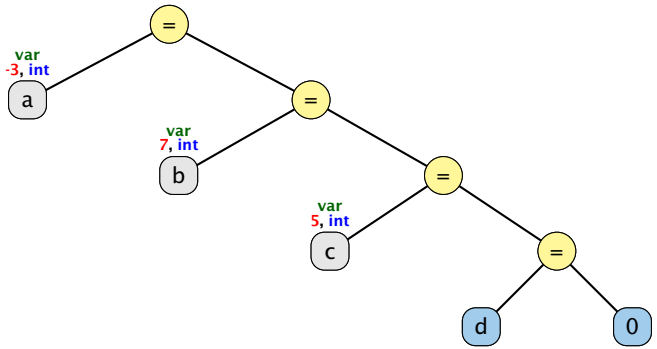
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



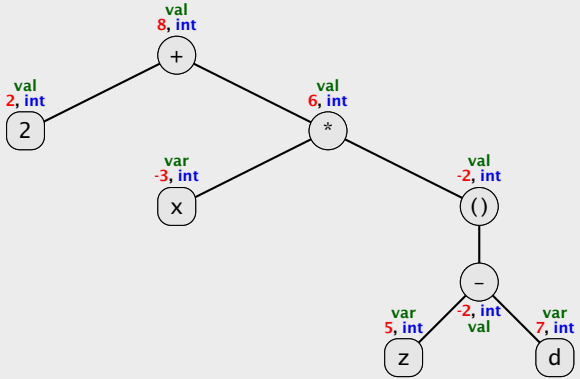
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



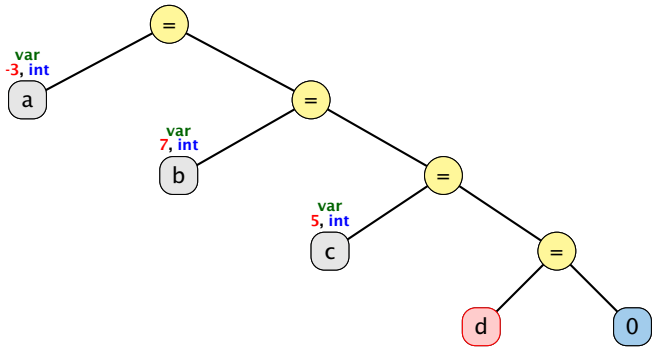
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



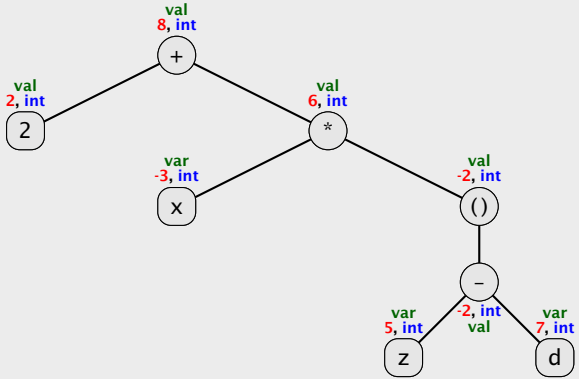
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



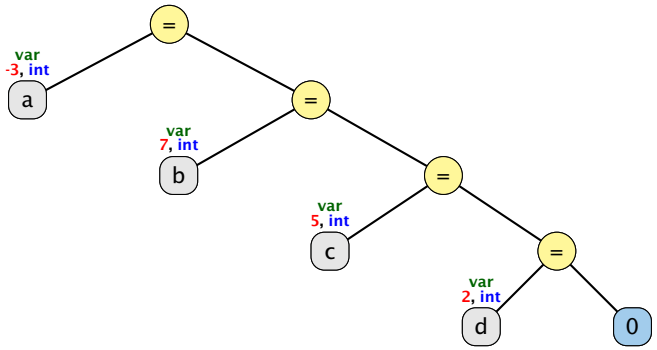
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



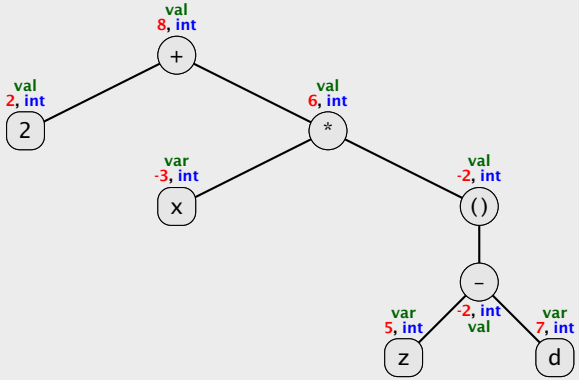
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



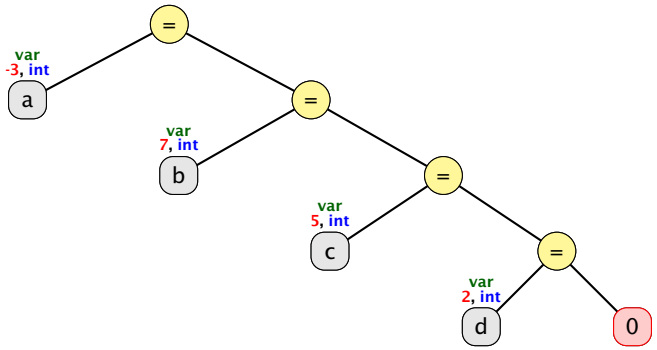
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



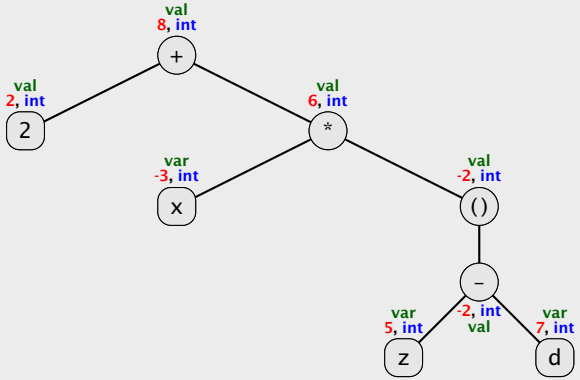
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



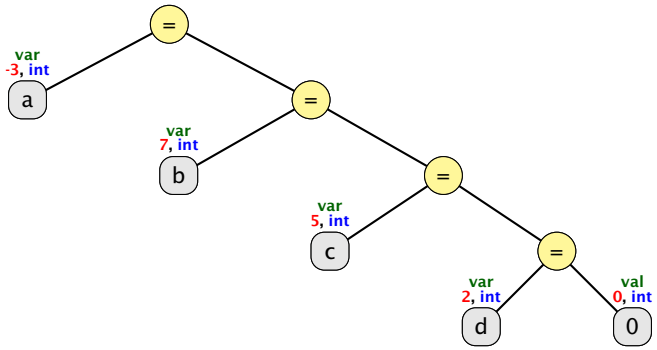
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



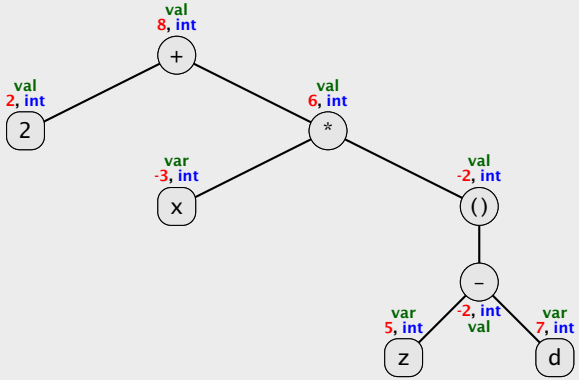
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



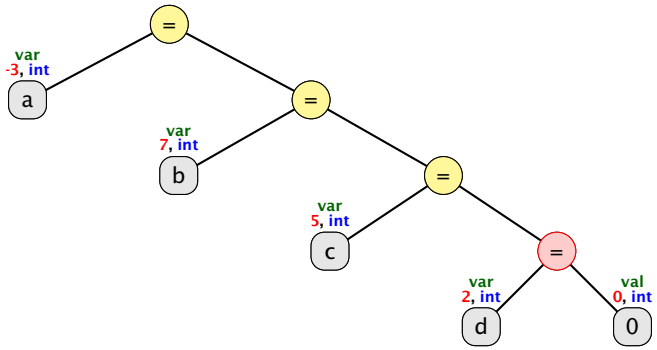
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



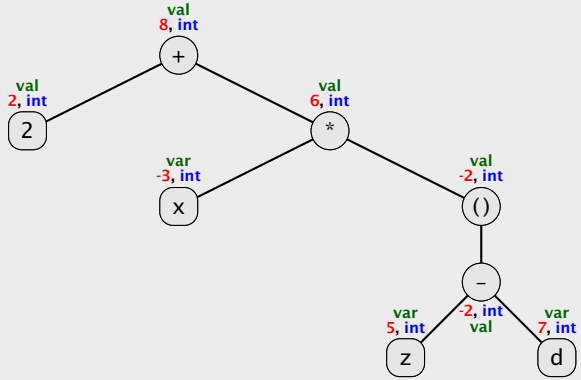
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



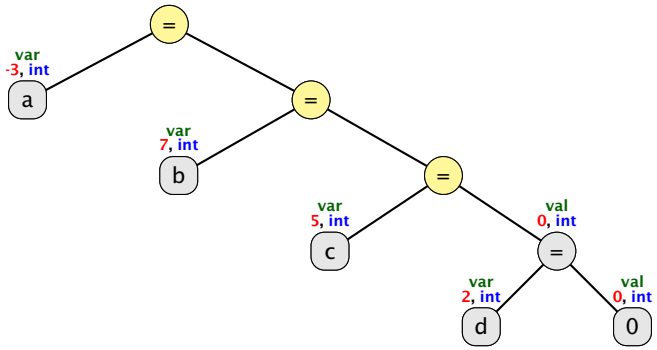
a -3 b 7 c 5 d 2

Beispiel: 2 + x * (z - d)



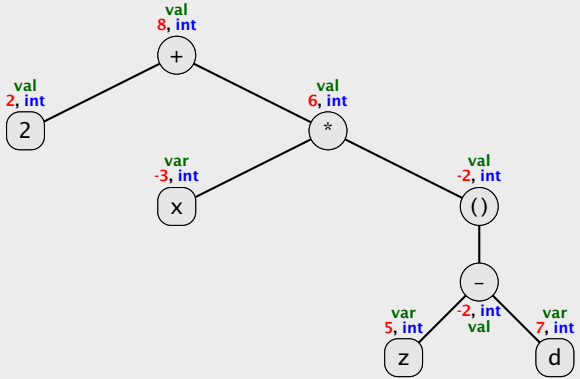
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



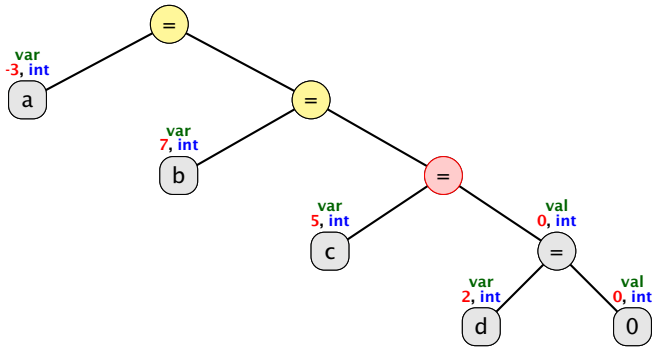
a -3 b 7 c 5 d 0

Beispiel: 2 + x * (z - d)



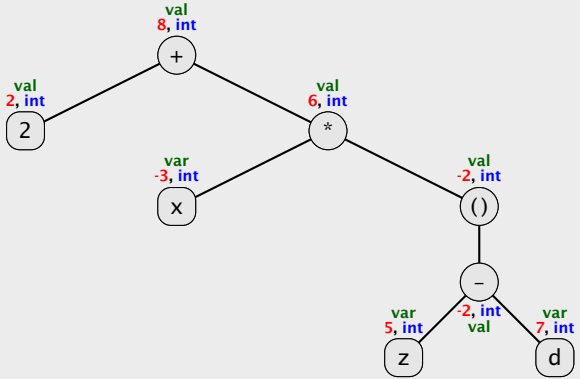
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



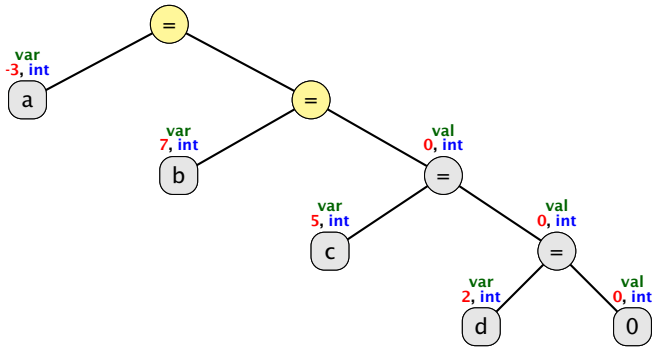
a -3 b 7 c 5 d 0

Beispiel: 2 + x * (z - d)



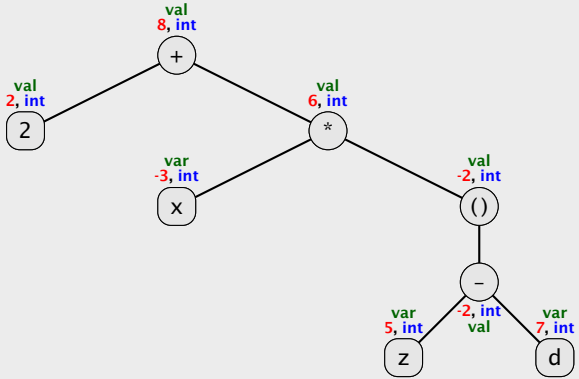
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



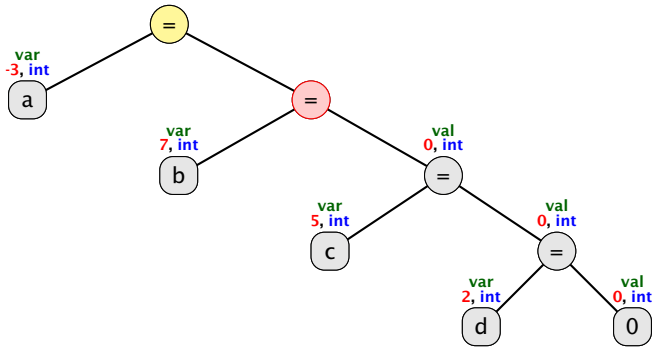
a -3 b 7 c 0 d 0

Beispiel: 2 + x * (z - d)



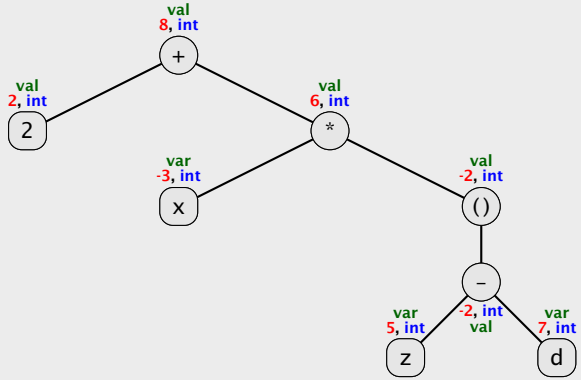
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



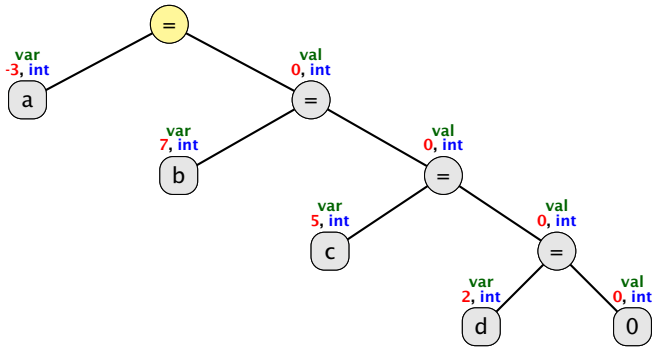
a -3 b 7 c 0 d 0

Beispiel: 2 + x * (z - d)



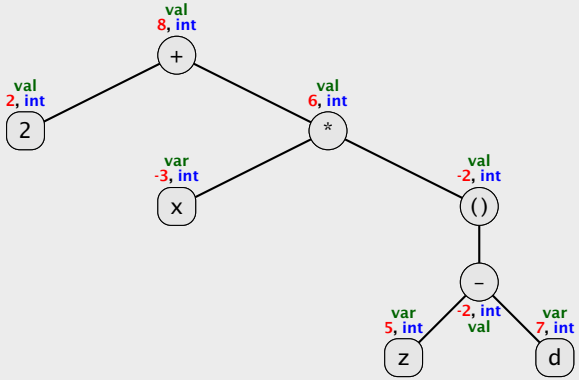
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



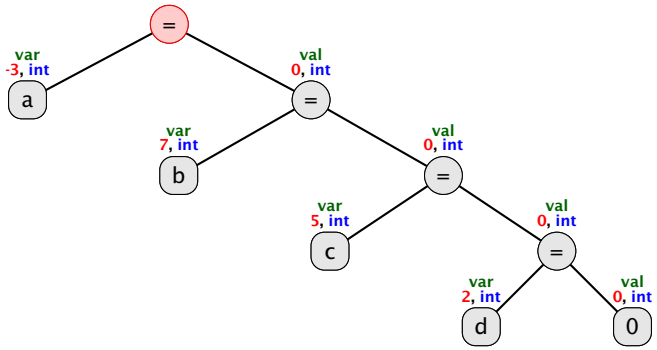
a -3 b 0 c 0 d 0

Beispiel: 2 + x * (z - d)



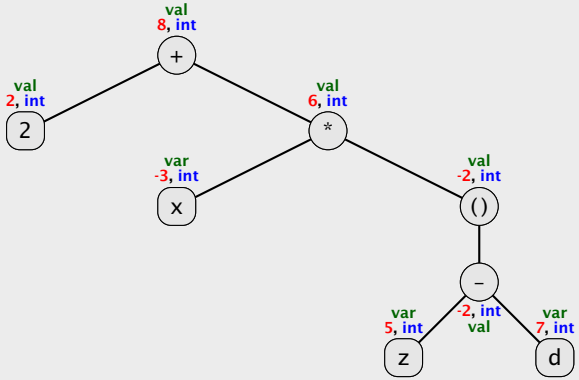
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



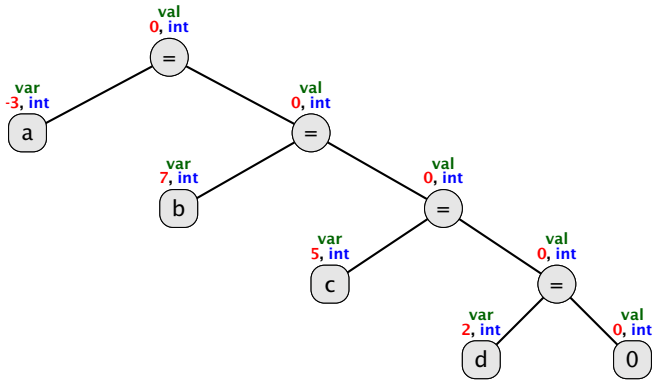
a -3 b 0 c 0 d 0

Beispiel: 2 + x * (z - d)



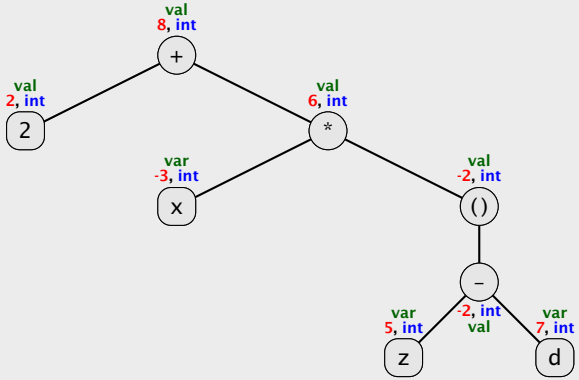
x -3 d 7 z 5

Beispiel: a = b = c = d = 0



a 0 b 0 c 0 d 0

Beispiel: 2 + x * (z - d)

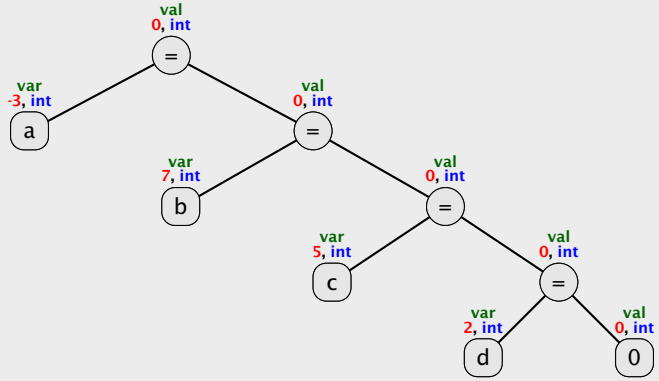


x -3 d 7 z 5

Beispiel: a != 0 && b/a < 10

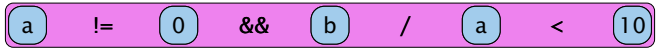
a	!=	0	&&	b	/	a	<	10

Beispiel: a = b = c = d = 0

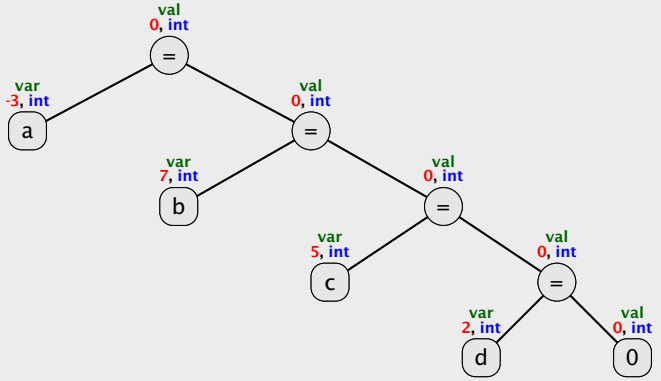


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

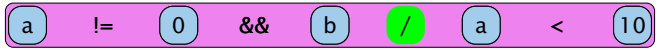


Beispiel: a = b = c = d = 0

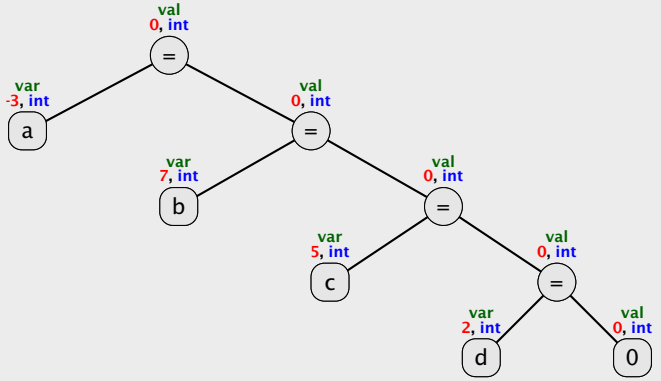


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

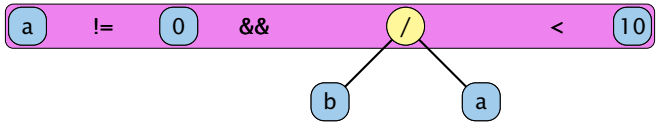


Beispiel: a = b = c = d = 0

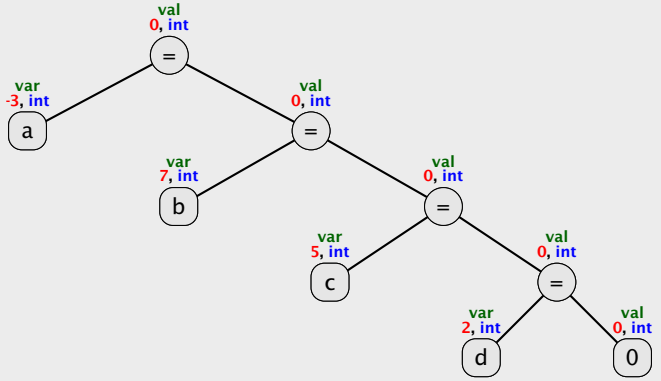


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

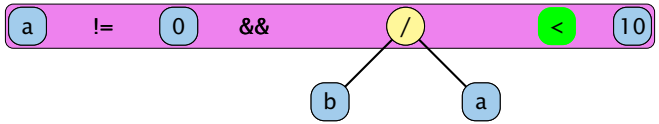


Beispiel: a = b = c = d = 0

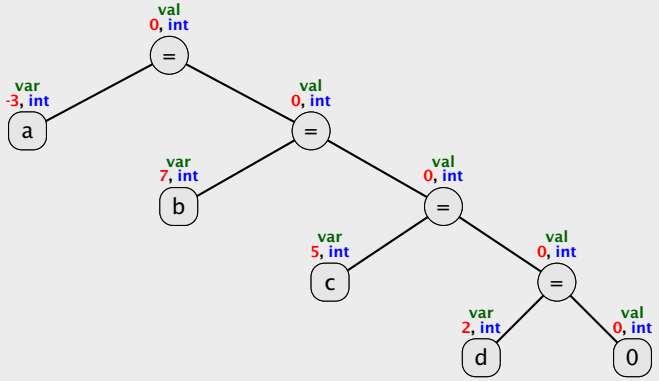


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

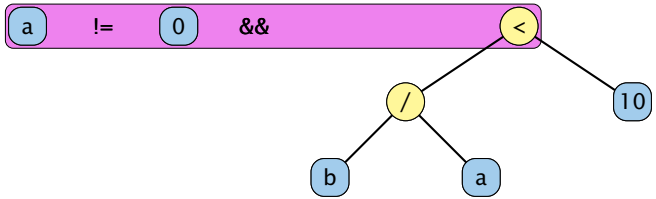


Beispiel: a = b = c = d = 0

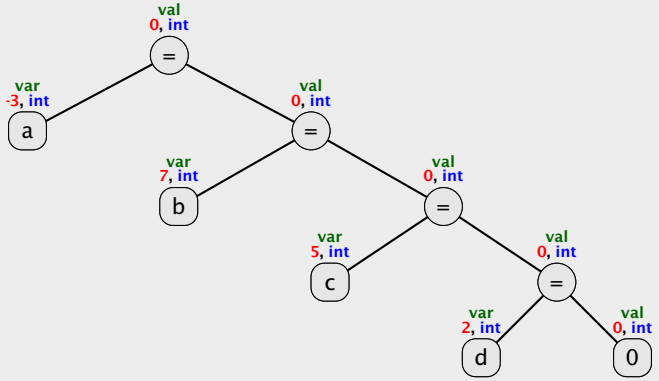


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

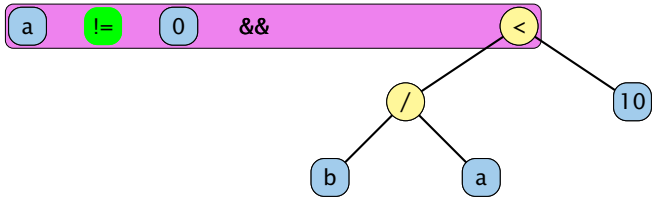


Beispiel: a = b = c = d = 0

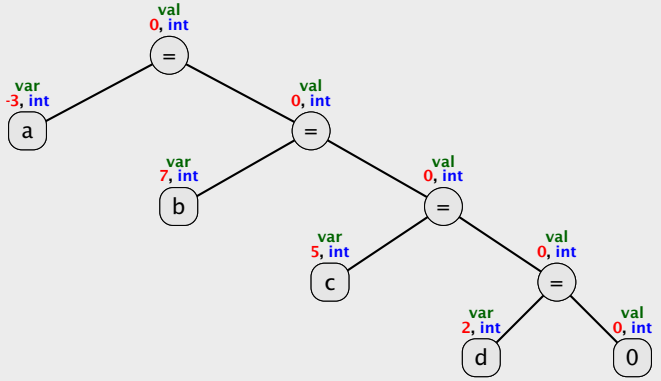


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

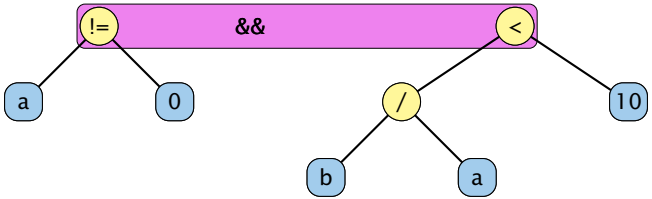


Beispiel: a = b = c = d = 0

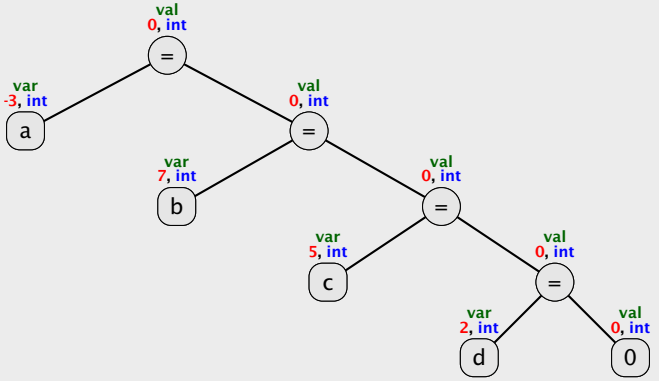


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

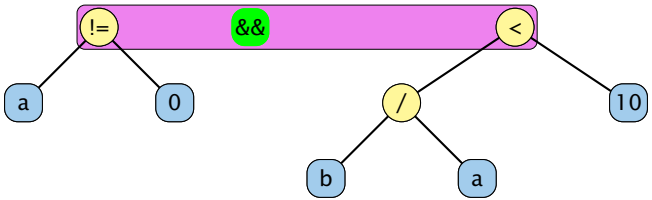


Beispiel: a = b = c = d = 0

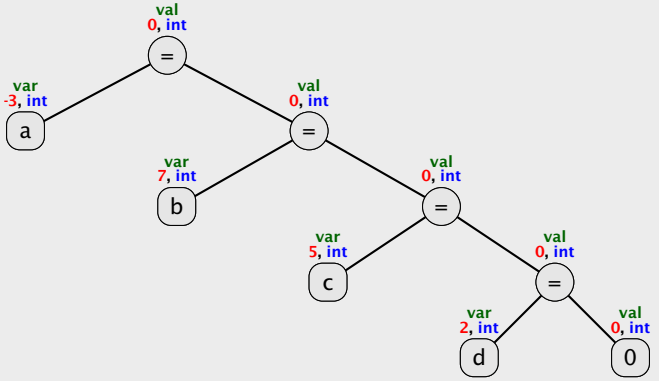


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

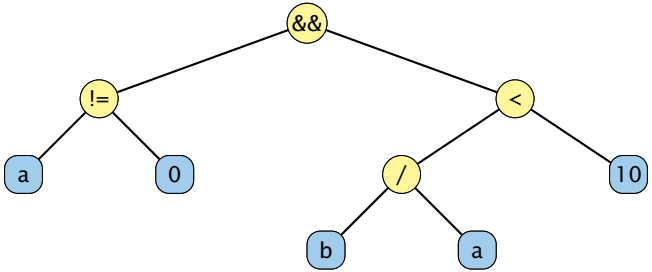


Beispiel: a = b = c = d = 0

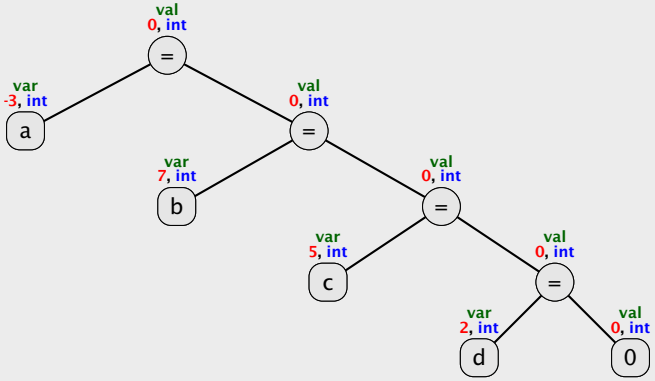


a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10

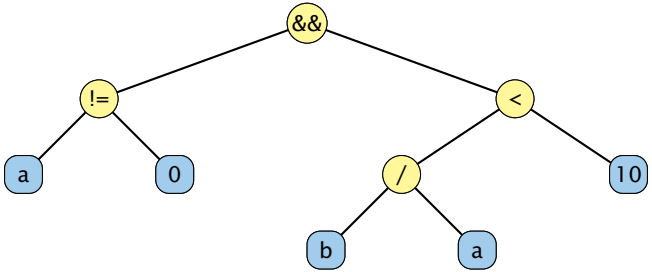


Beispiel: a = b = c = d = 0



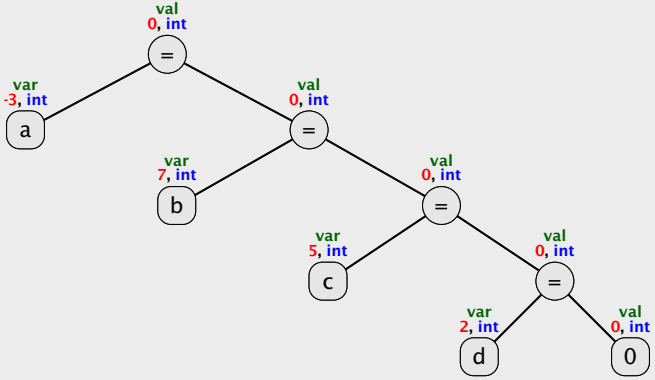
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



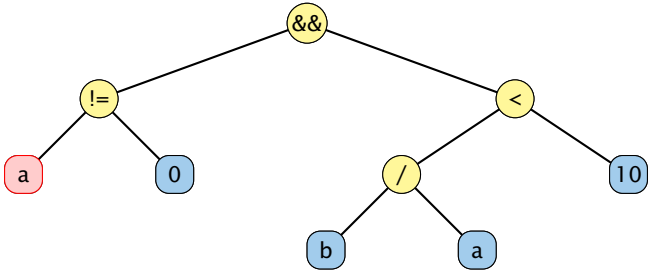
a 0 b 4

Beispiel: a = b = c = d = 0



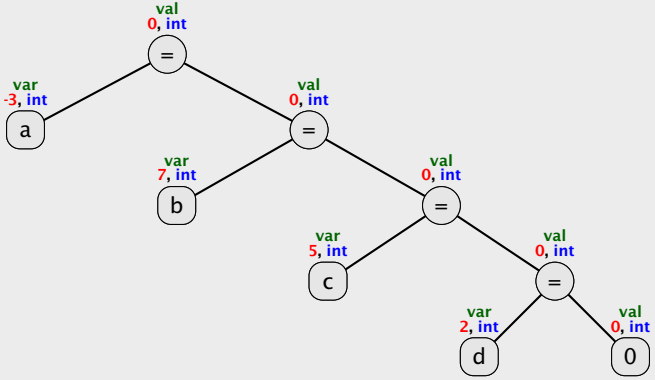
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



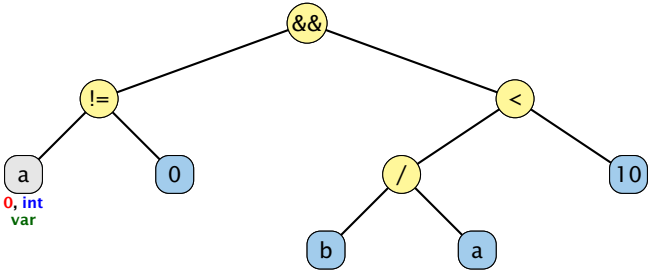
a 0 b 4

Beispiel: a = b = c = d = 0



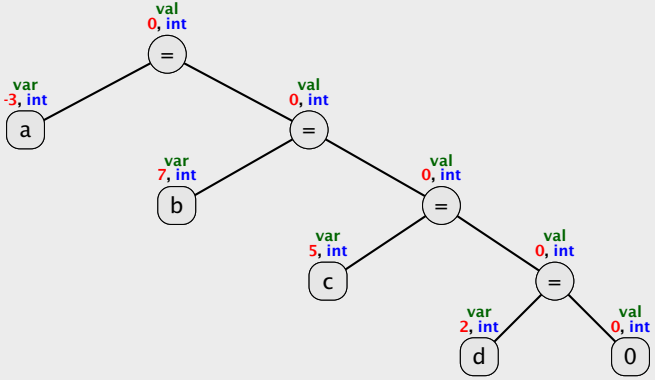
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



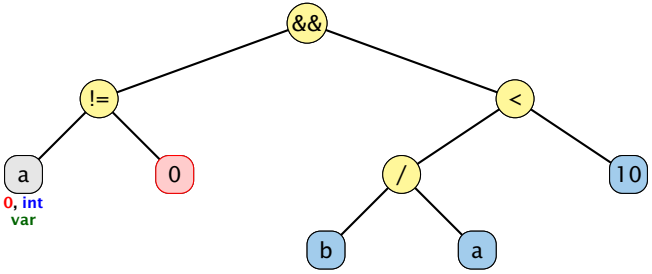
a 0 b 4

Beispiel: a = b = c = d = 0



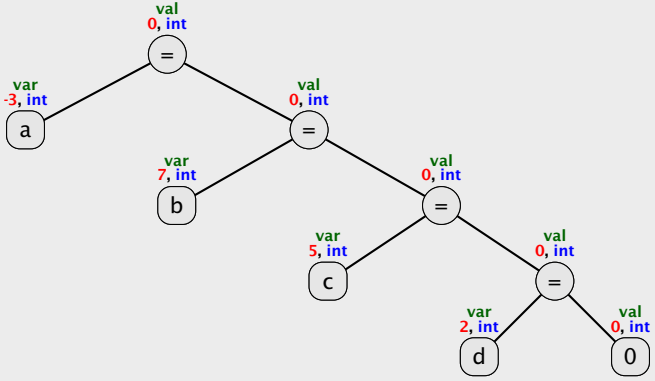
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



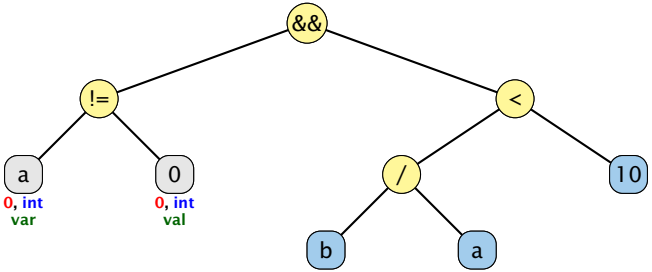
a 0 b 4

Beispiel: a = b = c = d = 0



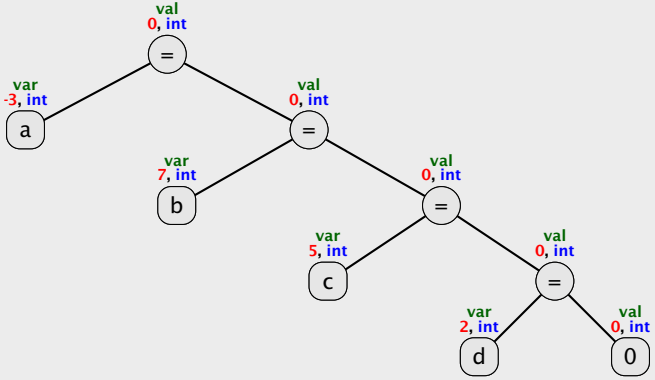
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



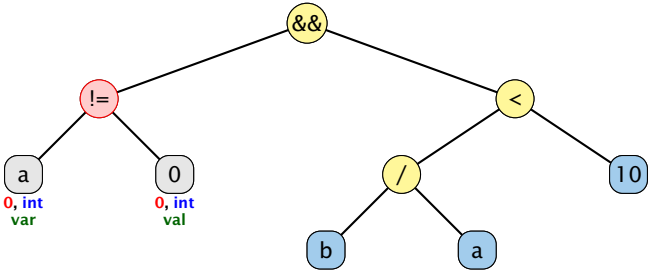
a 0 b 4

Beispiel: a = b = c = d = 0



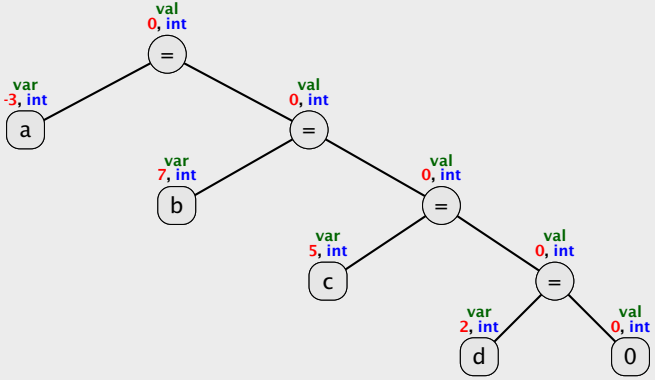
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



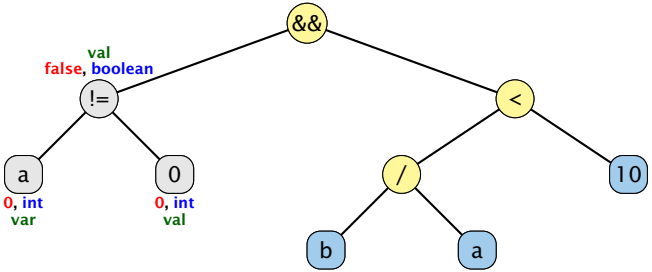
a 0 b 4

Beispiel: a = b = c = d = 0



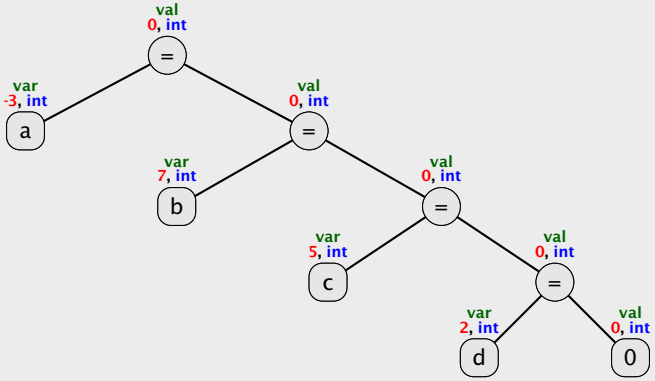
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



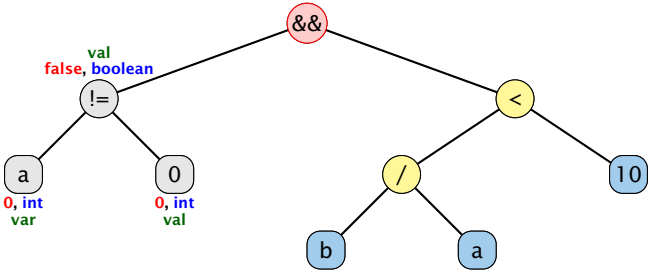
a 0 b 4

Beispiel: a = b = c = d = 0



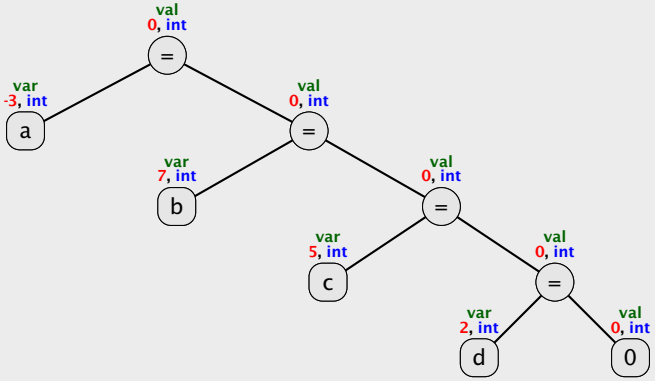
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



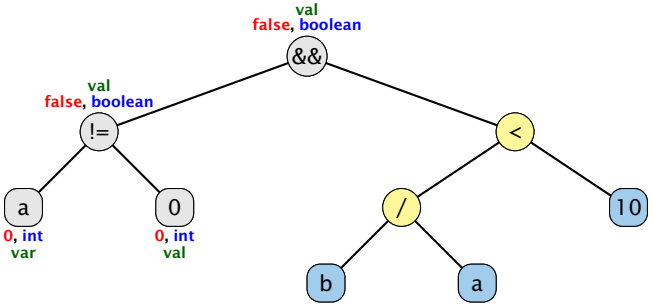
a 0 b 4

Beispiel: a = b = c = d = 0



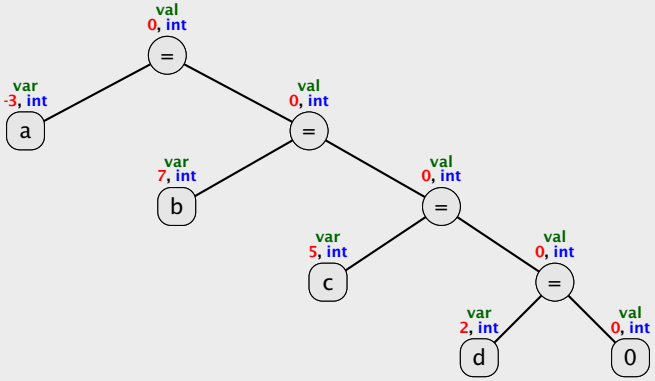
a 0 b 0 c 0 d 0

Beispiel: a != 0 && b/a < 10



a 0 b 4

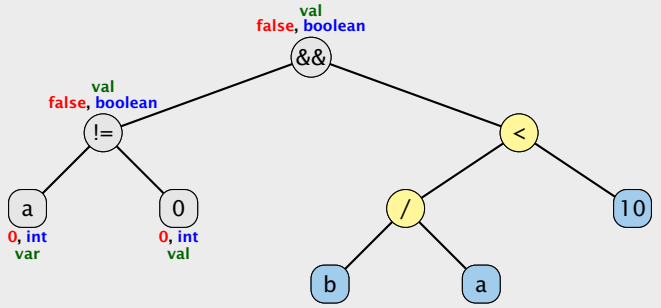
Beispiel: a = b = c = d = 0



a 0 b 0 c 0 d 0

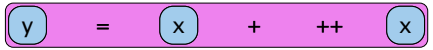
Beispiel: y = x + ++x

Beispiel: a != 0 && b/a < 10

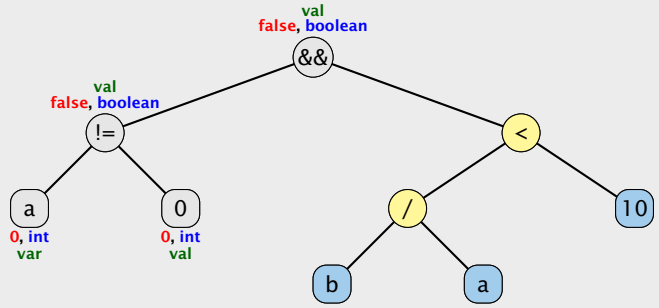


a 0 b 4

Beispiel: y = x + ++x

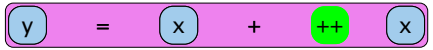


Beispiel: a != 0 && b/a < 10

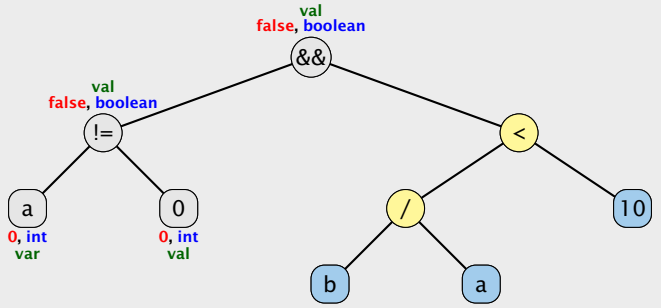


a 0 b 4

Beispiel: y = x + ++x

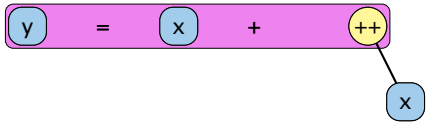


Beispiel: a != 0 && b/a < 10

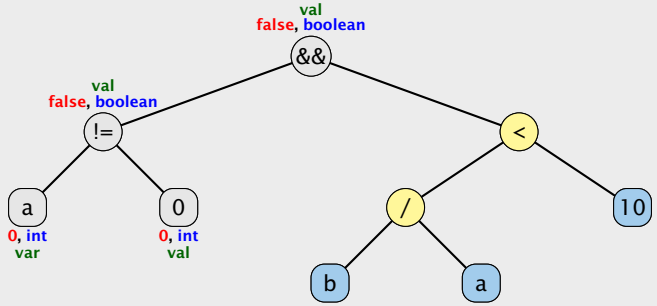


a 0 b 4

Beispiel: y = x + ++x

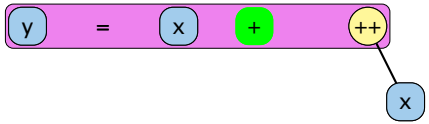


Beispiel: a != 0 && b/a < 10

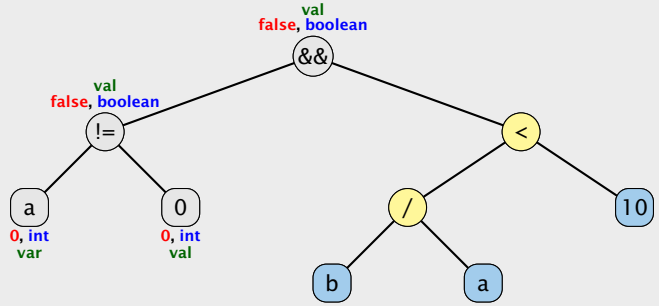


a 0 b 4

Beispiel: y = x + ++x

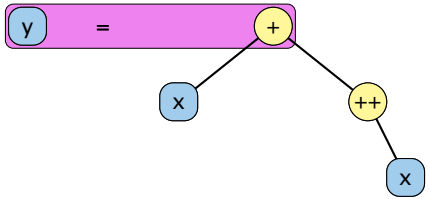


Beispiel: a != 0 && b/a < 10

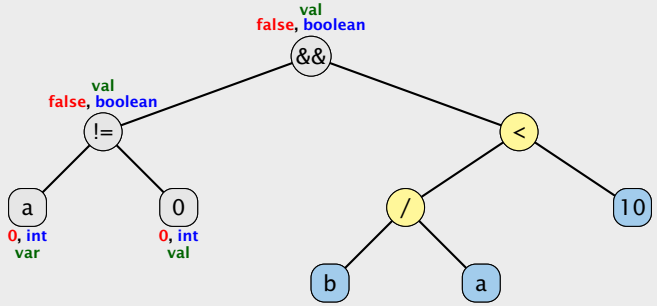


a 0 b 4

Beispiel: y = x + ++x

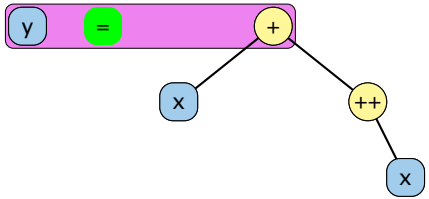


Beispiel: a != 0 && b/a < 10

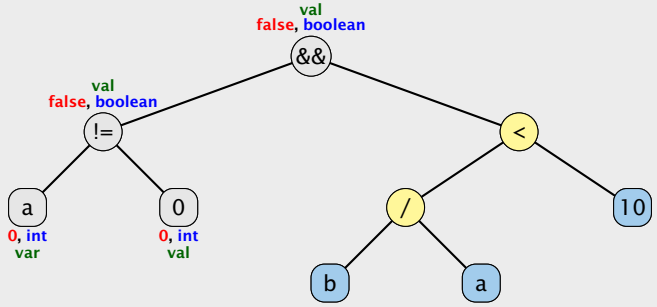


a 0 b 4

Beispiel: y = x + ++x

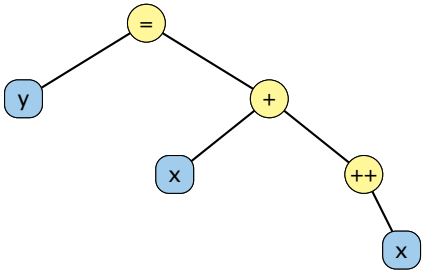


Beispiel: a != 0 && b/a < 10

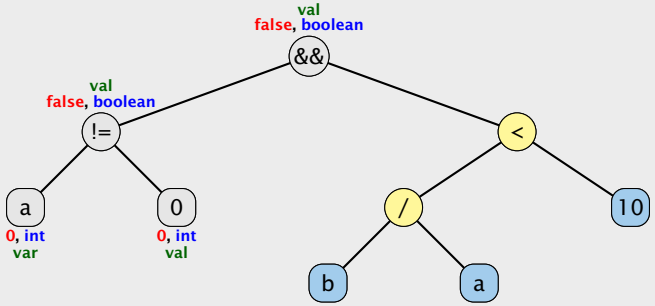


a 0 b 4

Beispiel: y = x + ++x

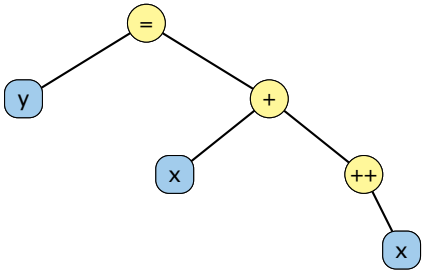


Beispiel: a != 0 && b/a < 10



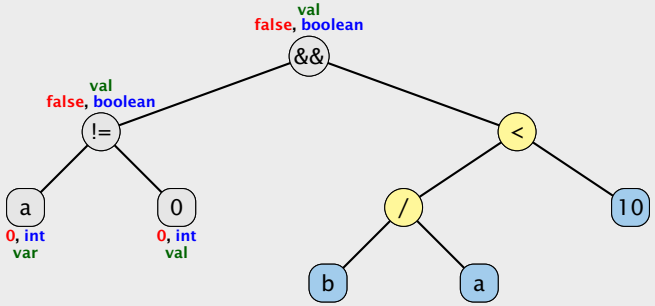
a 0 b 4

Beispiel: y = x + ++x



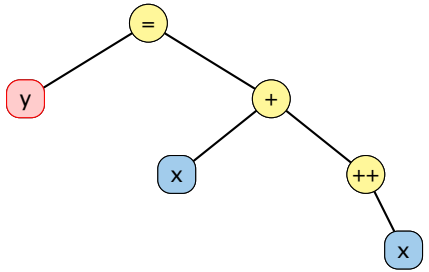
x 4 y 0

Beispiel: a != 0 && b/a < 10



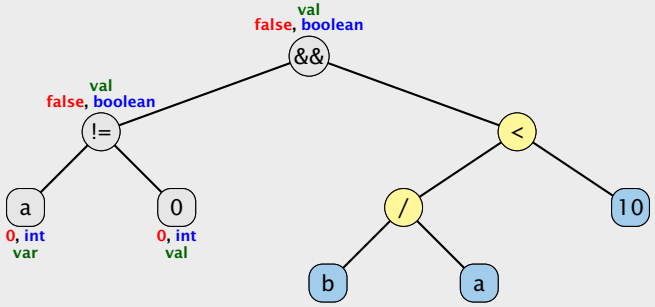
a 0 b 4

Beispiel: y = x + ++x



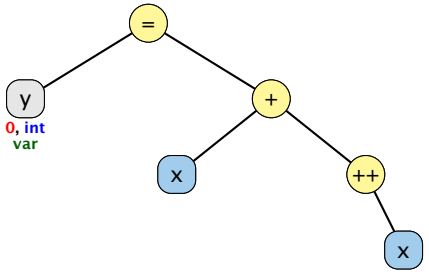
x 4 y 0

Beispiel: a != 0 && b/a < 10



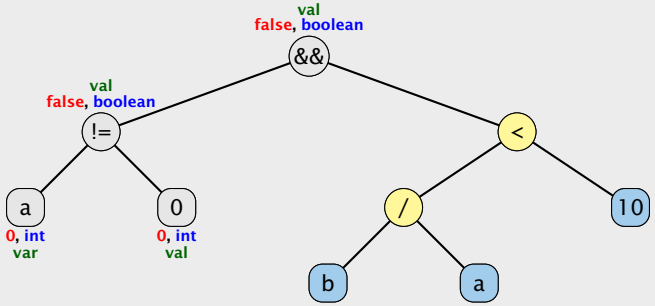
a 0 b 4

Beispiel: y = x + ++x



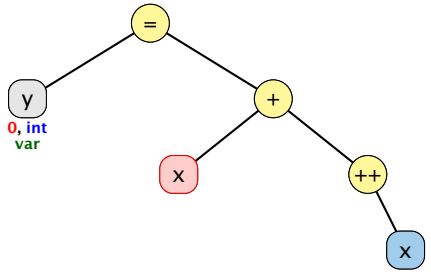
x 4 y 0

Beispiel: a != 0 && b/a < 10



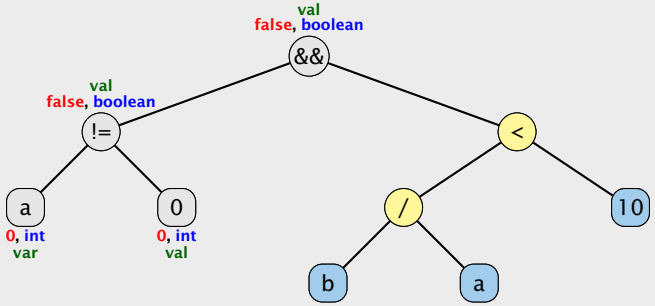
a 0 b 4

Beispiel: y = x + ++x



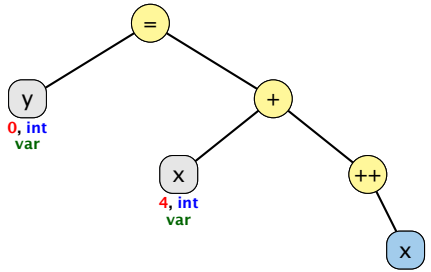
x 4 y 0

Beispiel: a != 0 && b/a < 10



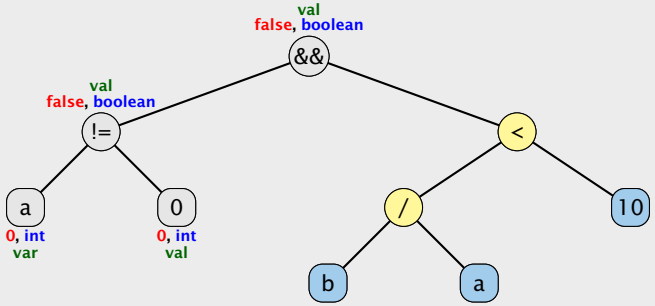
a 0 b 4

Beispiel: y = x + ++x



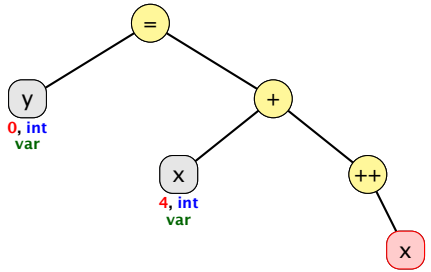
x 4 y 0

Beispiel: a != 0 && b/a < 10



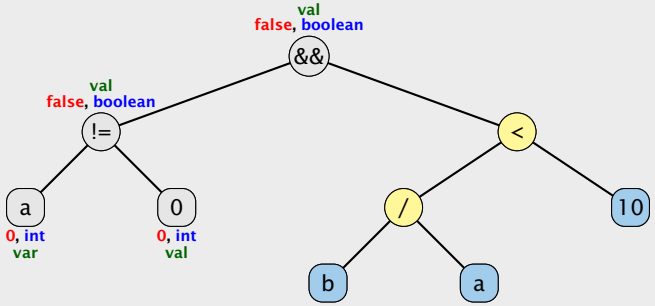
a 0 b 4

Beispiel: y = x + ++x



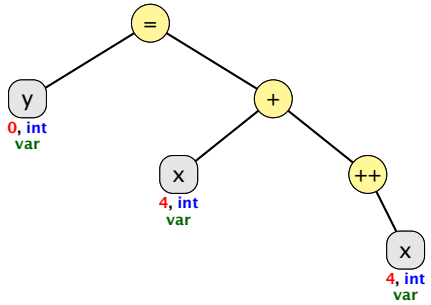
x 4 y 0

Beispiel: a != 0 && b/a < 10



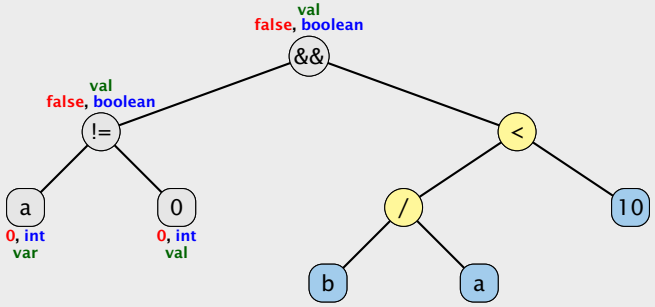
a 0 b 4

Beispiel: y = x + ++x



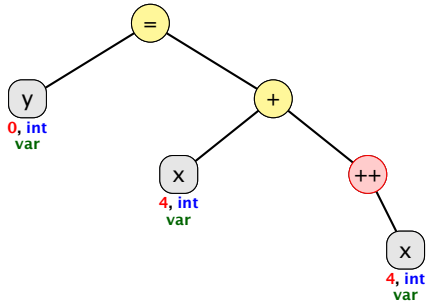
x 4 y 0

Beispiel: a != 0 && b/a < 10



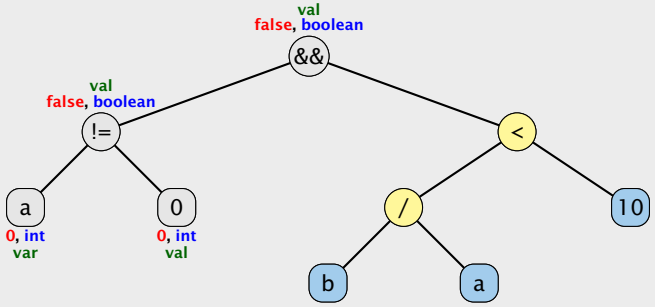
a 0 b 4

Beispiel: y = x + ++x



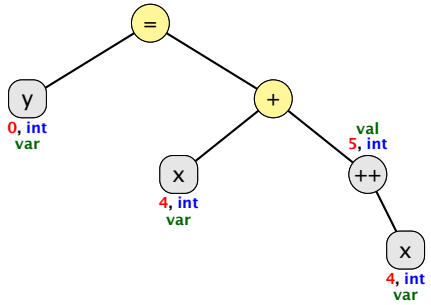
x 4 y 0

Beispiel: a != 0 && b/a < 10



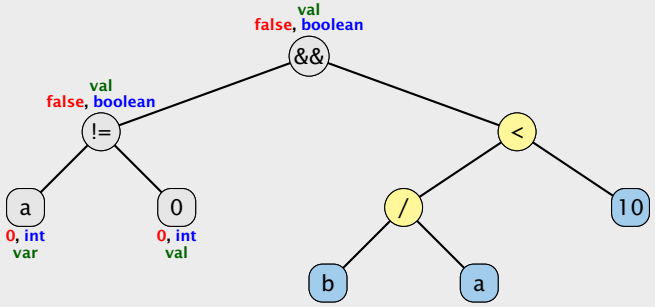
a 0 b 4

Beispiel: y = x + ++x



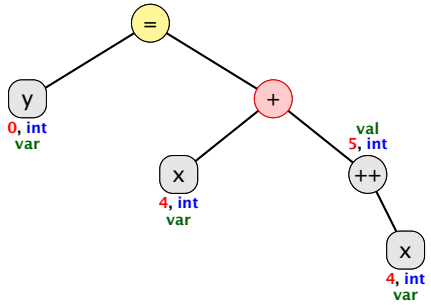
x 5 y 0

Beispiel: a != 0 && b/a < 10



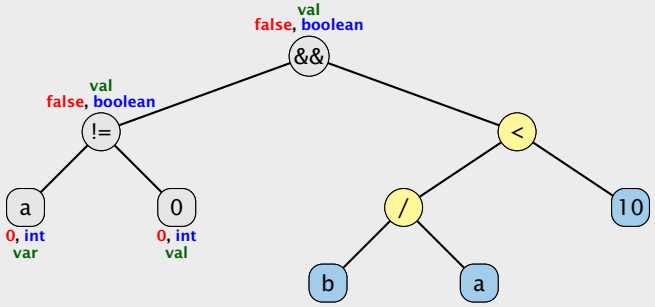
a 0 b 4

Beispiel: y = x + ++x



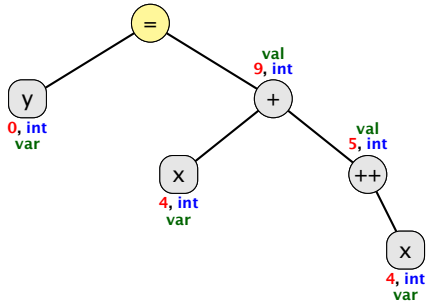
x 5 y 0

Beispiel: a != 0 && b/a < 10



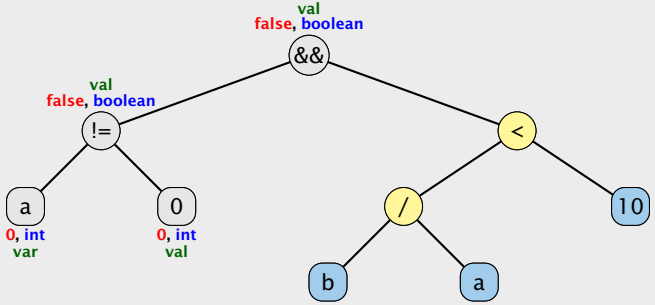
a 0 b 4

Beispiel: y = x + ++x



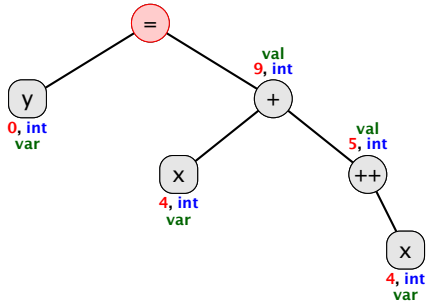
x 5 y 0

Beispiel: a != 0 && b/a < 10



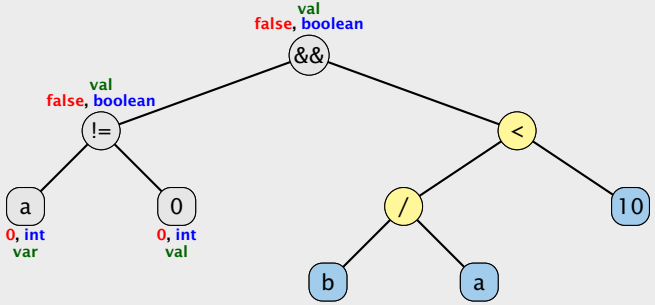
a 0 b 4

Beispiel: y = x + ++x



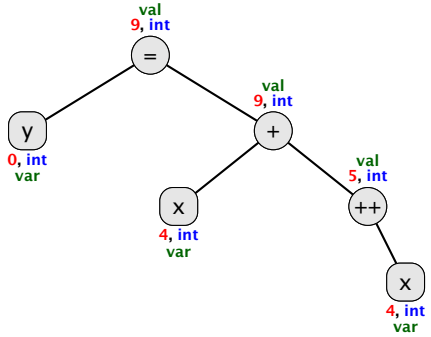
x 5 y 0

Beispiel: a != 0 && b/a < 10



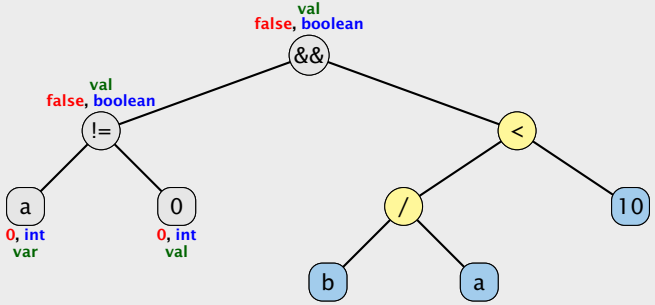
a 0 b 4

Beispiel: y = x + ++x



x 5 y 9

Beispiel: a != 0 && b/a < 10

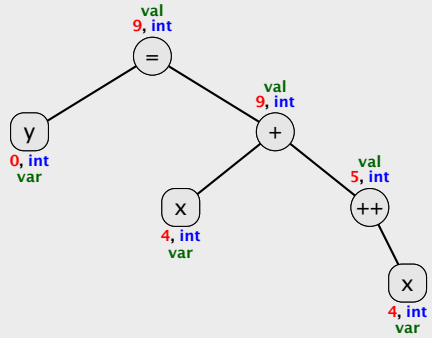


a 0 b 4

Beispiel: y = x++ + x

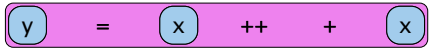
y	=	x	++	+	x

Beispiel: y = x + ++x

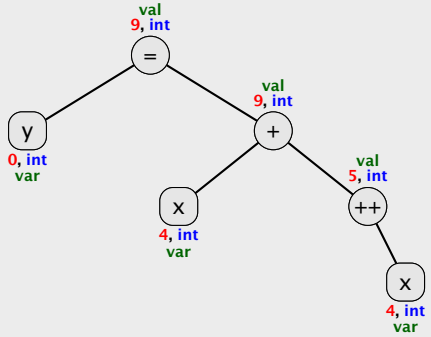


x 5 y 9

Beispiel: y = x++ + x



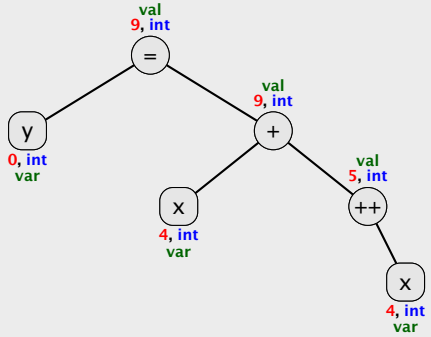
Beispiel: y = x + ++x



x 5 y 9

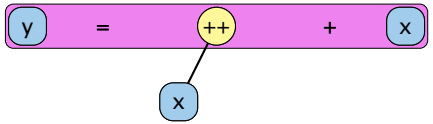
Beispiel: y = x++ + x

Beispiel: y = x + ++x

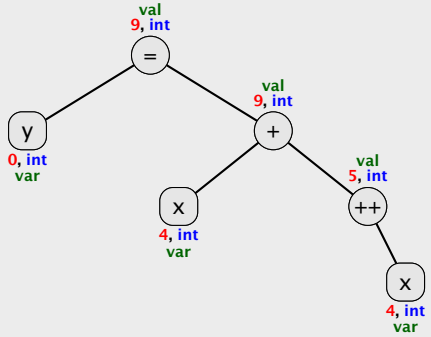


x 5 y 9

Beispiel: y = x++ + x

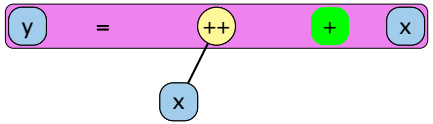


Beispiel: y = x + ++x

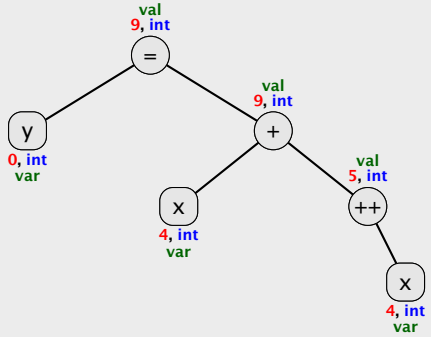


x 5 y 9

Beispiel: y = x++ + x

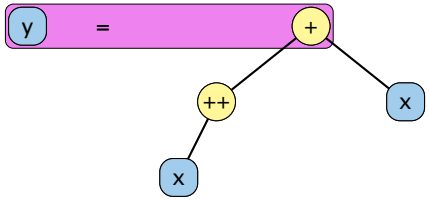


Beispiel: y = x + ++x

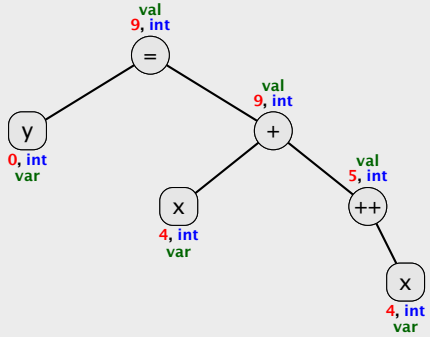


x 5 y 9

Beispiel: y = x++ + x

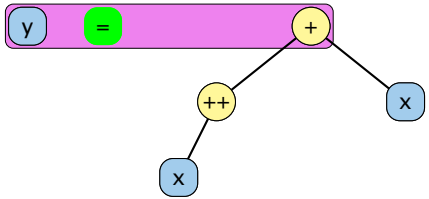


Beispiel: y = x + ++x

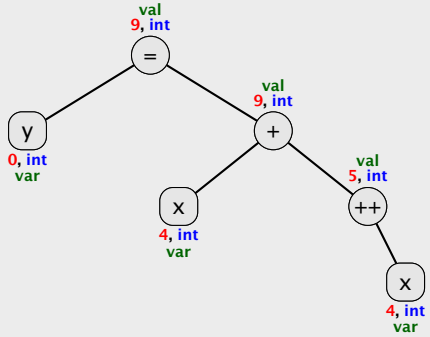


x 5 y 9

Beispiel: y = x++ + x

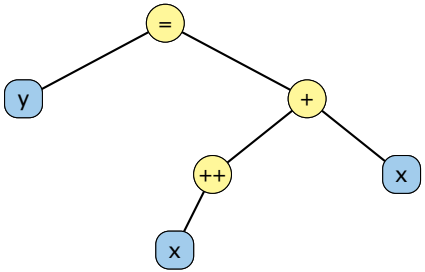


Beispiel: y = x + ++x

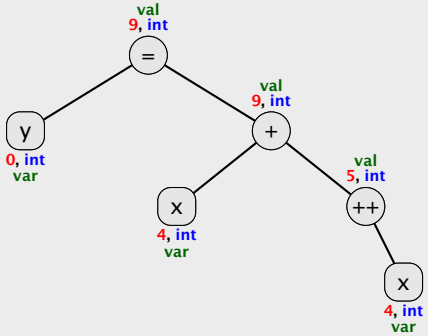


x 5 y 9

Beispiel: y = x++ + x

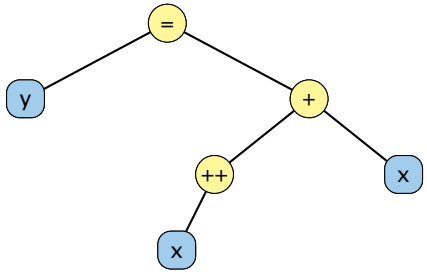


Beispiel: y = x + ++x



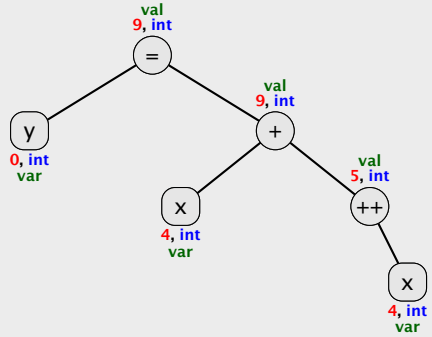
x 5 y 9

Beispiel: y = x++ + x



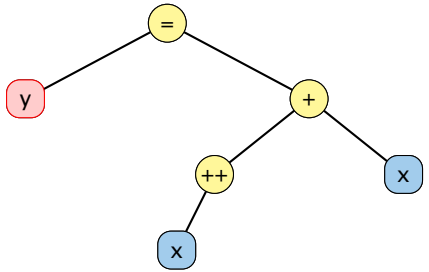
x 4 y 0

Beispiel: y = x + ++x



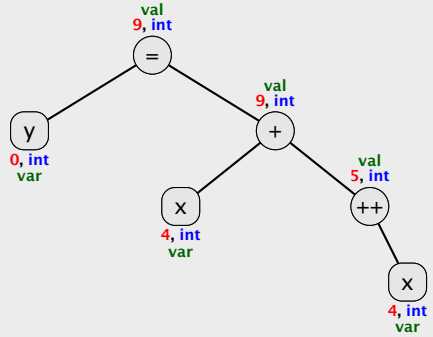
x 5 y 9

Beispiel: y = x++ + x



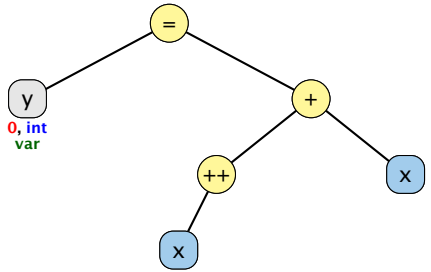
x 4 y 0

Beispiel: y = x + ++x



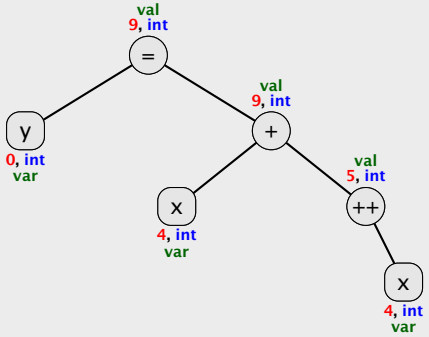
x 5 y 9

Beispiel: y = x++ + x



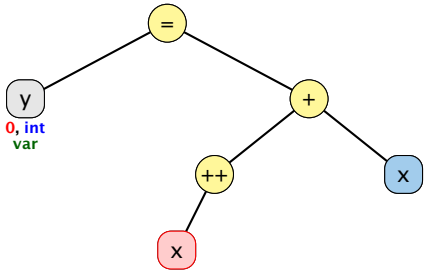
x 4 y 0

Beispiel: y = x + ++x



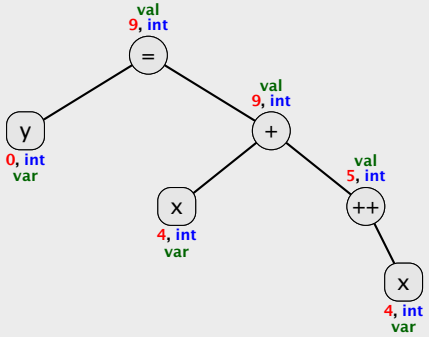
x 5 y 9

Beispiel: y = x++ + x



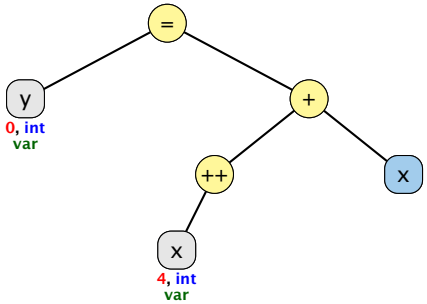
x 4 y 0

Beispiel: y = x + ++x



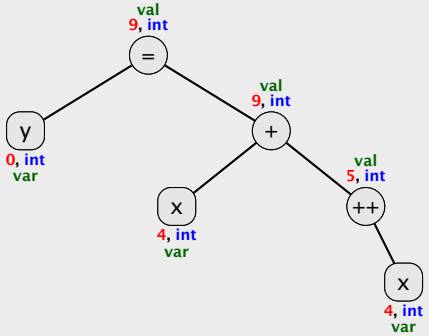
x 5 y 9

Beispiel: y = x++ + x



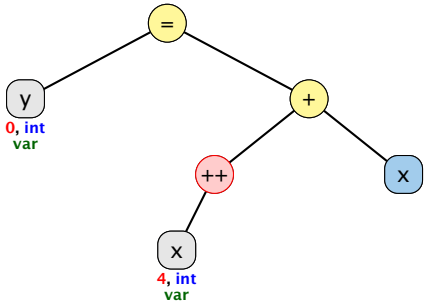
x 4 y 0

Beispiel: y = x + ++x



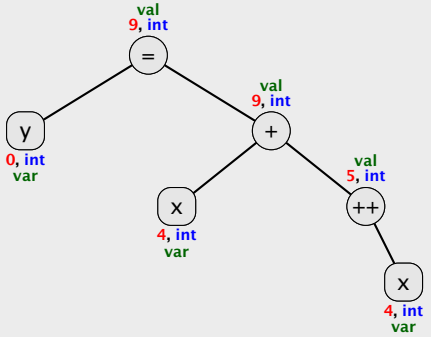
x 5 y 9

Beispiel: y = x++ + x



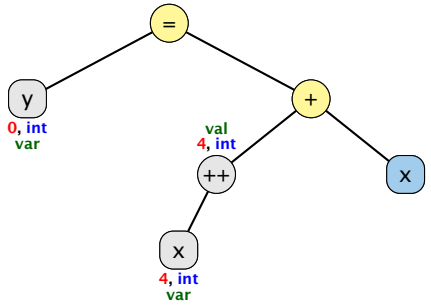
x 4 y 0

Beispiel: y = x + ++x



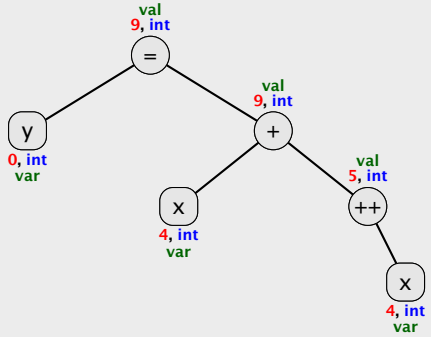
x 5 y 9

Beispiel: y = x++ + x



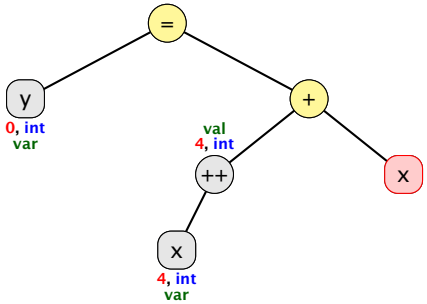
x 5 y 0

Beispiel: y = x + ++x



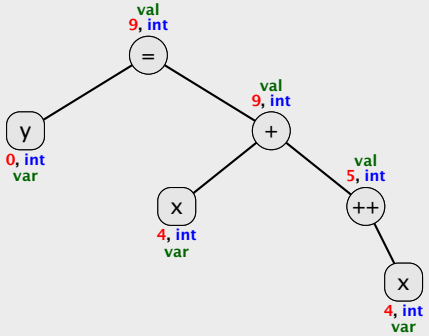
x 5 y 9

Beispiel: y = x++ + x



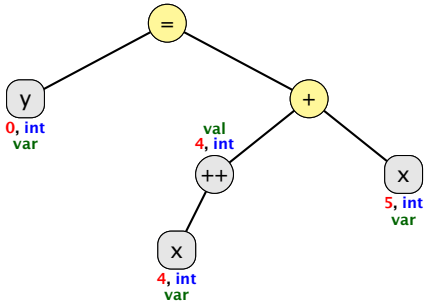
x 5 y 0

Beispiel: y = x + ++x



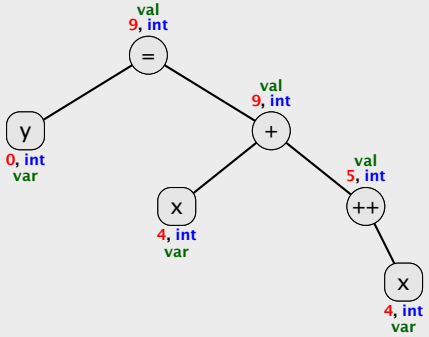
x 5 y 9

Beispiel: y = x++ + x



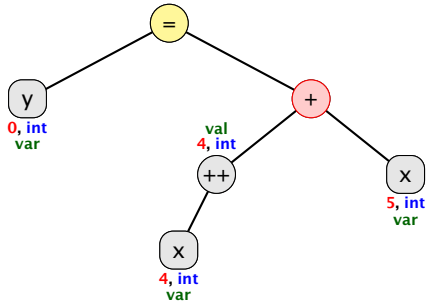
x 5 y 0

Beispiel: y = x + ++x



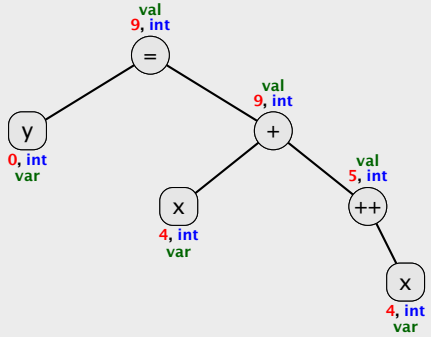
x 5 y 9

Beispiel: y = x++ + x



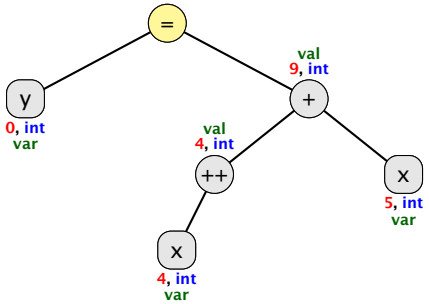
x 5 y 0

Beispiel: y = x + ++x



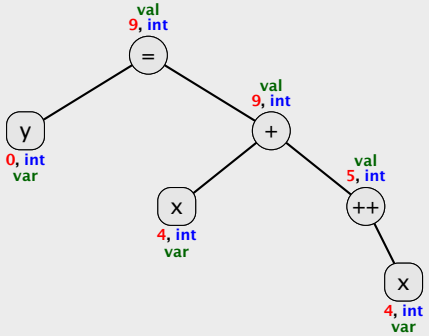
x 5 y 9

Beispiel: y = x++ + x



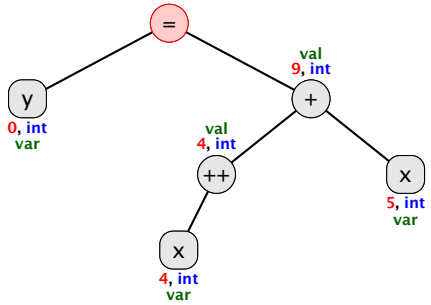
x 5 y 0

Beispiel: y = x + ++x



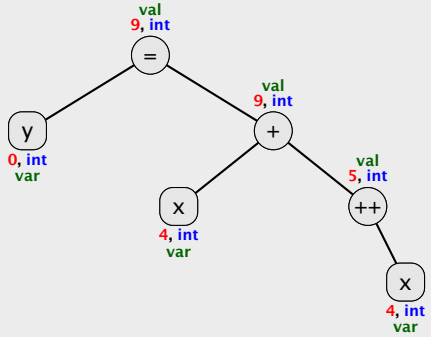
x 5 y 9

Beispiel: y = x++ + x



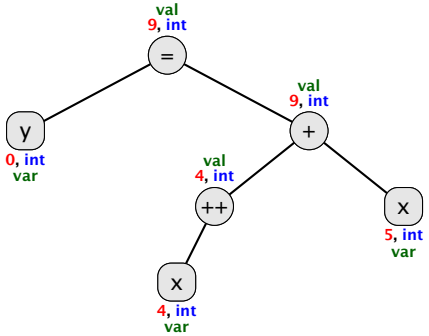
x 5 y 0

Beispiel: y = x + ++x



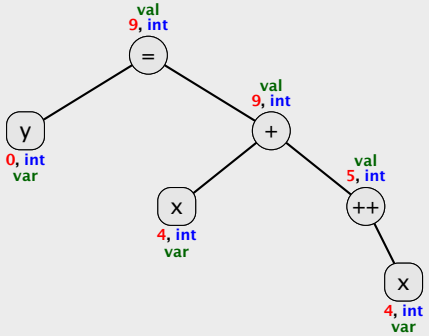
x 5 y 9

Beispiel: y = x++ + x



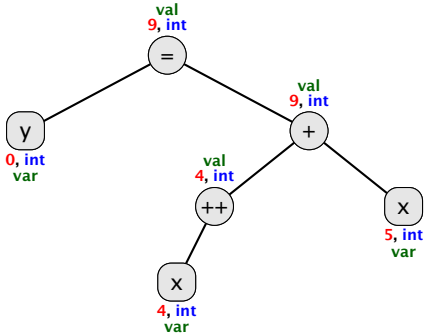
x 5 y 9

Beispiel: y = x + ++x



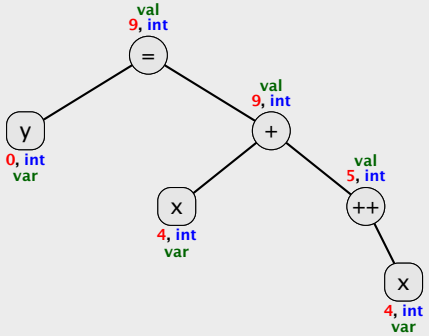
x 5 y 9

Beispiel: y = x++ + x



x 5 y 9

Beispiel: y = x + ++x



x 5 y 9

Impliziter Typecast

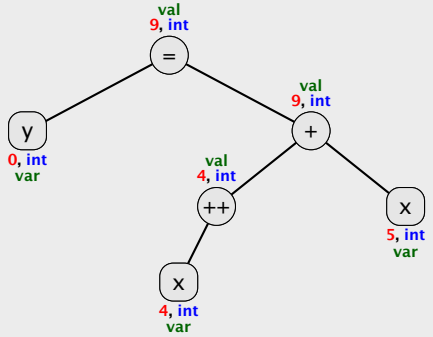
Wenn ein Ausdruck vom TypA an einer Stelle verwendet wird, wo ein Ausdruck vom TypB erforderlich ist, wird

- entweder der Ausdruck vom TypA in einen Ausdruck vom TypB gecasted (impliziter Typecast),
- oder ein Compilerfehler erzeugt, falls dieser Cast nicht (automatisch) erlaubt ist.

Beispiel: Zuweisung

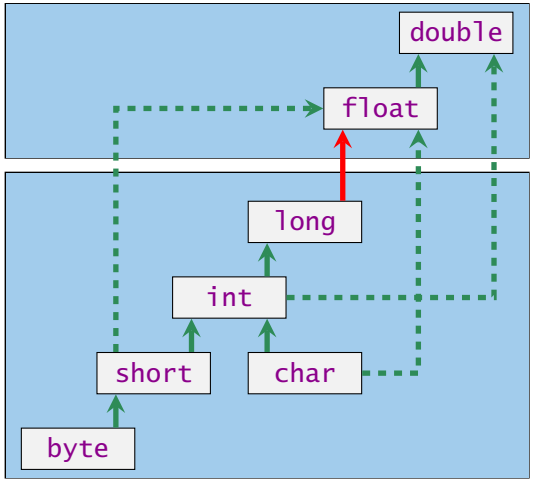
```
long x = 5;
int  y = 3;
x = y; // impliziter Cast von int nach long
```

Beispiel: y = x++ + x



x 5 y 9

Erlaubte Implizite Typecasts – Numerische Typen



Konvertierung von `long` nach `double` oder von `int` nach `float` kann Information verlieren wird aber automatisch durchgeführt.

Impliziter Typecast

Wenn ein Ausdruck vom TypA an einer Stelle verwendet wird, wo ein Ausdruck vom TypB erforderlich ist, wird

- entweder der Ausdruck vom TypA in einen Ausdruck vom TypB gecasted (impliziter Typecast),
- oder ein Compilerfehler erzeugt, falls dieser Cast nicht (automatisch) erlaubt ist.

Beispiel: Zuweisung

```
long x = 5;
int  y = 3;
x = y; // impliziter Cast von int nach long
```

Welcher Typ wird benötigt?

Operatoren sind üblicherweise überladen, d.h. ein Symbol (+, -, ...) steht in Abhängigkeit der Parameter (Argumente) für unterschiedliche Funktionen.

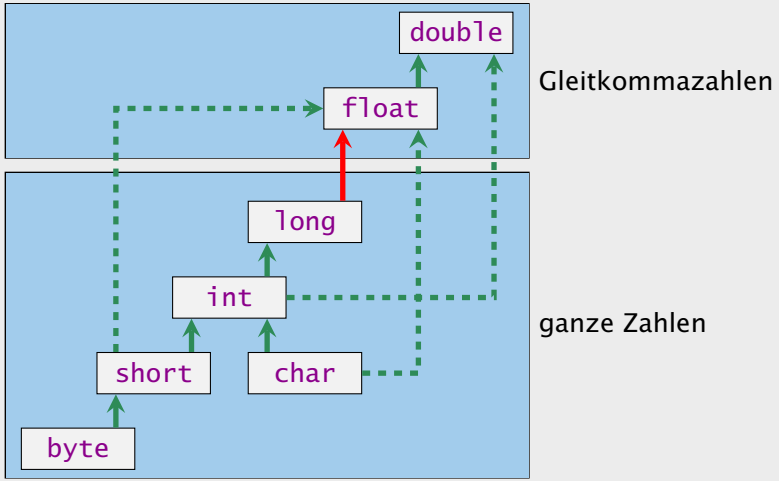
```
+ : int ⟶ int
+ : long ⟶ long
+ : float ⟶ float
+ : double ⟶ double

+ : int × int ⟶ int
+ : long × long ⟶ long
+ : float × float ⟶ float
+ : double × double ⟶ double

+ : String × String ⟶ String
```

Der Compiler muss in der Lage sein während der Compilierung die richtige Funktion zu bestimmen.

Erlaubte Implizite Typecasts – Numerische Typen



Konvertierung von `long` nach `double` oder von `int` nach `float` kann Information verlieren wird aber automatisch durchgeführt.

Impliziter Typecast

Der Compiler wertet nur die Typen des Ausdrucksbaums aus.

- Für jeden inneren Knoten wählt er dann die geeignete Funktion (z.B. `+ : long` $\times$ `long` $\longrightarrow$ `long`) falls ein +-Knoten zwei `long`-Argumente erhält.
- Falls keine passende Funktion gefunden wird, versucht der Compiler durch implizite Typecasts die Operanden an eine Funktion anzupassen.
- Dies geschieht auch für selbstgeschriebene Funktionen (z.B. `min(int a, int b)` und `min(long a, long b)`).
- Der Compiler nimmt die Funktion mit der speziellsten Signatur.

Welcher Typ wird benötigt?

Operatoren sind üblicherweise überladen, d.h. ein Symbol (+, -, ...) steht in Abhängigkeit der Parameter (Argumente) für unterschiedliche Funktionen.

```
+ : int ⟶ int
+ : long ⟶ long
+ : float ⟶ float
+ : double ⟶ double

+ : int × int ⟶ int
+ : long × long ⟶ long
+ : float × float ⟶ float
+ : double × double ⟶ double

+ : String × String ⟶ String
```

Der Compiler muss in der Lage sein während der Compilierung die richtige Funktion zu bestimmen.

Speziellste Signatur

Impliziter Typecast

Der Compiler wertet nur die Typen des Ausdrucksbaums aus.

- Für jeden inneren Knoten wählt er dann die geeignete Funktion (z.B. `+ : long` $\times$ `long` $\longrightarrow$ `long`) falls ein +-Knoten zwei `long`-Argumente erhält.
- Falls keine passende Funktion gefunden wird, versucht der Compiler durch implizite Typecasts die Operanden an eine Funktion anzupassen.
- Dies geschieht auch für selbstgeschriebene Funktionen (z.B. `min(int a, int b)` und `min(long a, long b)`).
- Der Compiler nimmt die Funktion mit der speziellsten Signatur.

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T'_1, \ldots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \ldots, k\}$.

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf der Menge der k-Tupel von Typen.

Speziellste Signatur

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

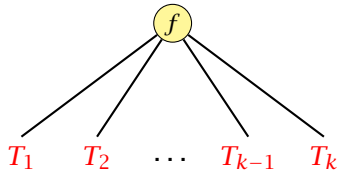
Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T_1', \ldots, T_k')$ falls $T_i \preceq T_i'$ für alle $i \in \{1, \ldots, k\}$.

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf der Menge der k-Tupel von Typen.

Speziellste Signatur

$R_1 \quad f(\mathcal{T}_1)$
$R_2 \quad f(\mathcal{T}_2)$
$\vdots$
$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ sind k-Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \ldots, T_k)$ ist das k-tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i | \mathcal{T} \preceq_k \mathcal{T}_i\} \ .$$

Wähle kleinstes Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

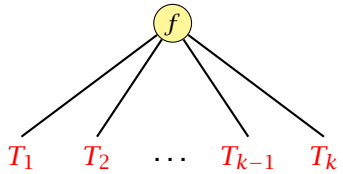
d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T'_1, \ldots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \ldots, k\}$.

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf der Menge der k-Tupel von Typen.

$R_1 \quad f(\mathcal{T}_1)$
$R_2 \quad f(\mathcal{T}_2)$
$\vdots$
$R_\ell \quad f(\mathcal{T}_\ell)$



$\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ sind k-Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \ldots, T_k)$ ist das k-tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i | \mathcal{T} \preceq_k \mathcal{T}_i\} \ .$$

Wähle kleinstes Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

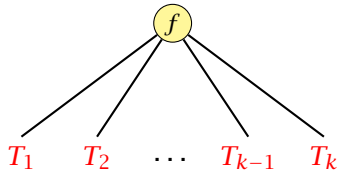
d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T'_1, \ldots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \ldots, k\}$.

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf der Menge der k-Tupel von Typen.

$$\begin{array}{ll} R_1 & f(\mathcal{T}_1) \\ R_2 & f(\mathcal{T}_2) \\ & \vdots \\ R_\ell & f(\mathcal{T}_\ell) \end{array}$$



$\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ sind k-Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \ldots, T_k)$ ist das k-tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i | \mathcal{T} \preceq_k \mathcal{T}_i\} \ .$$

Wähle kleinstes Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Ordnungsrelationen

Relation $\preceq$: TypA $\preceq$ TypB falls TypA nach TypB (implizit) gecasted werden kann:

- **reflexiv:** $T \preceq T$
- **transitiv:** $T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow T_1 \preceq T_3$
- **antisymmetrisch:** $T_1 \preceq T_2 \wedge T_2 \preceq T_1 \Rightarrow T_1 = T_2$

d.h., $\preceq$ definiert Halbordnung auf der Menge der Typen.

Relation $\preceq_k$: $(T_1, \ldots, T_k) \preceq_k (T'_1, \ldots, T'_k)$ falls $T_i \preceq T'_i$ für alle $i \in \{1, \ldots, k\}$.

- **reflexiv:** $\mathcal{T} \preceq_k \mathcal{T}$
- **transitiv:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_3 \Rightarrow \mathcal{T}_1 \preceq_k \mathcal{T}_3$
- **antisymmetrisch:** $\mathcal{T}_1 \preceq_k \mathcal{T}_2 \wedge \mathcal{T}_2 \preceq_k \mathcal{T}_1 \Rightarrow \mathcal{T}_1 = \mathcal{T}_2$

d.h., $\preceq_k$ definiert Halbordnung auf der Menge der k-Tupel von Typen.

Impliziter Typecast – Numerische Typen

Angenommen wir haben Funktionen

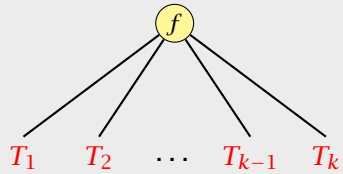
```
int min(int a, int b)
float min(float a, float b)
double min(double a, double b)
```

definiert.

```
1 long a = 7, b = 3;
2 double d = min(a, b);
```

würde die Funktion `float min(float a, float b)` aufrufen.

$$\begin{array}{ll} R_1 & f(\mathcal{T}_1) \\ R_2 & f(\mathcal{T}_2) \\ & \vdots \\ R_\ell & f(\mathcal{T}_\ell) \end{array}$$



$\mathcal{T}_1, \ldots, \mathcal{T}_\ell$ sind k-Tupel von Typen für die eine Definition von f existiert.

$\mathcal{T} = (T_1, \ldots, T_k)$ ist das k-tupel von Typen mit dem f aufgerufen wird.

Menge aller möglichen Funktionen/Tupel:

$$M := \{\mathcal{T}_i | \mathcal{T} \preceq_k \mathcal{T}_i\} \ .$$

Wähle kleinstes Element aus M falls M ein eindeutig kleinstes Element besitzt (sonst Compilerfehler).

Impliziter Typecast – Numerische Typen

Angenommen wir haben Funktionen

```
int min(int a, int b)
float min(float a, float b)
double min(double a, double b)
```

definiert.

```
1 long a = 7, b = 3;
2 double d = min(a, b);
```

würde die Funktion `float min(float a, float b)` aufrufen.

Impliziter Typecast

Bei Ausdrücken mit Seiteneffekten (Zuweisungen, ++ , --) gelten andere Regeln:

Beispiel: Zuweisungen

$= : \texttt{byte}^* \times \texttt{byte} \longrightarrow \texttt{byte}$
$= : \texttt{char}^* \times \texttt{char} \longrightarrow \texttt{char}$
$= : \texttt{short}^* \times \texttt{short} \longrightarrow \texttt{short}$
$= : \texttt{int}^* \times \texttt{int} \longrightarrow \texttt{int}$
$= : \texttt{long}^* \times \texttt{long} \longrightarrow \texttt{long}$
$= : \texttt{float}^* \times \texttt{float} \longrightarrow \texttt{float}$
$= : \texttt{double}^* \times \texttt{double} \longrightarrow \texttt{double}$

Es wird nur der Parameter konvertiert, der nicht dem Seiteneffekt unterliegt.

Impliziter Typecast

Bei Ausdrücken mit Seiteneffekten (Zuweisungen, ++ , --) gelten andere Regeln:

Beispiel: Zuweisungen

$= : \texttt{byte}^* \times \texttt{byte} \longrightarrow \texttt{byte}$
$= : \texttt{char}^* \times \texttt{char} \longrightarrow \texttt{char}$
$= : \texttt{short}^* \times \texttt{short} \longrightarrow \texttt{short}$
$= : \texttt{int}^* \times \texttt{int} \longrightarrow \texttt{int}$
$= : \texttt{long}^* \times \texttt{long} \longrightarrow \texttt{long}$
$= : \texttt{float}^* \times \texttt{float} \longrightarrow \texttt{float}$
$= : \texttt{double}^* \times \texttt{double} \longrightarrow \texttt{double}$

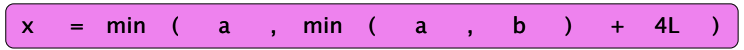
Es wird nur der Parameter konvertiert, der nicht dem Seiteneffekt unterliegt.

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
`()`	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)



5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

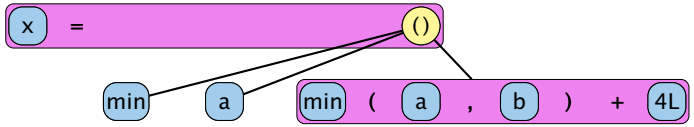
Beispiel: x = min(a, min(a,b) + 4L)

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

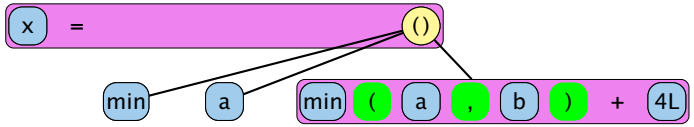


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

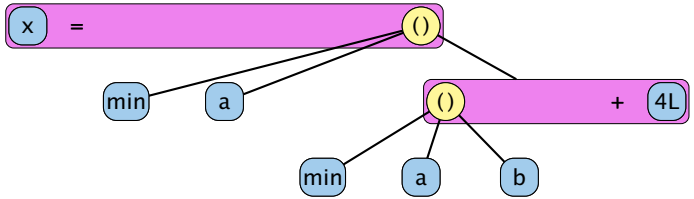


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

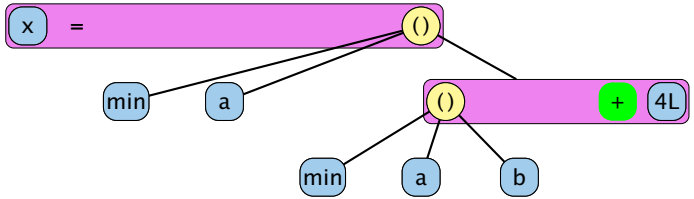


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

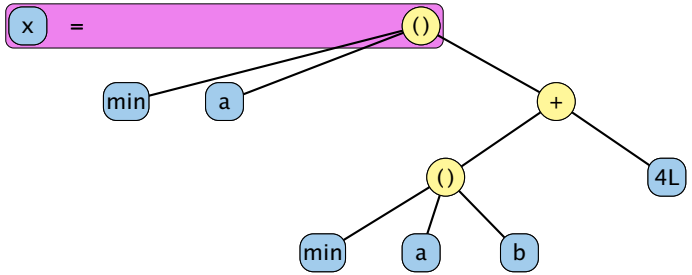


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

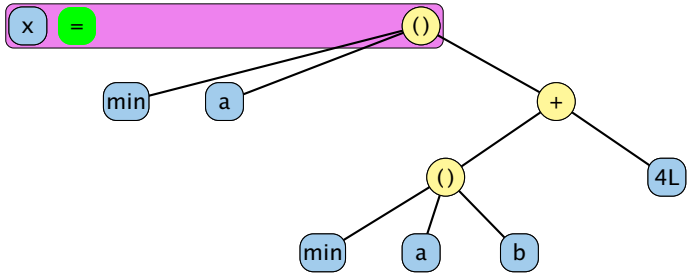


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

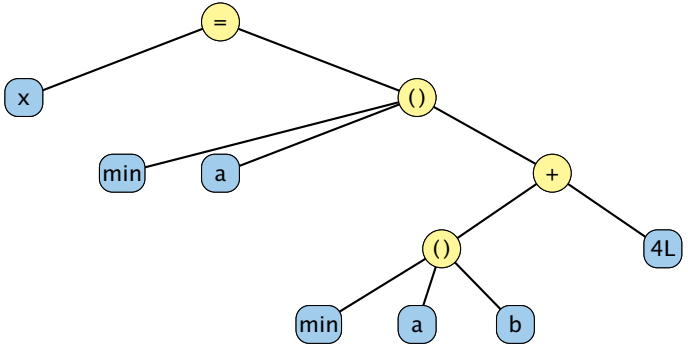


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)




```
int min(int,int)
float min(float,float)
double min(double,double)
```

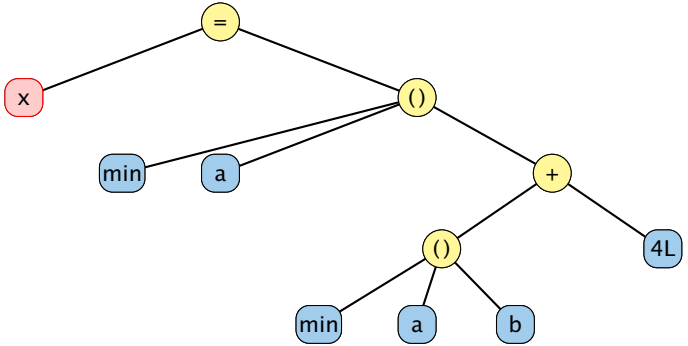
long x 7 int a 3 int b 5

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)




```
int min(int,int)
float min(float,float)
double min(double,double)
```

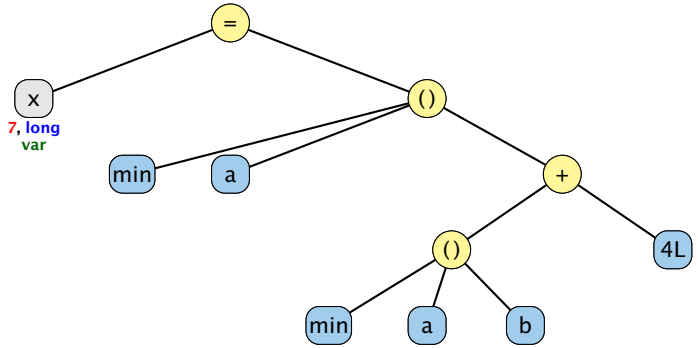
long x 7 int a 3 int b 5

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)




```
int min(int,int)
float min(float,float)
double min(double,double)
```

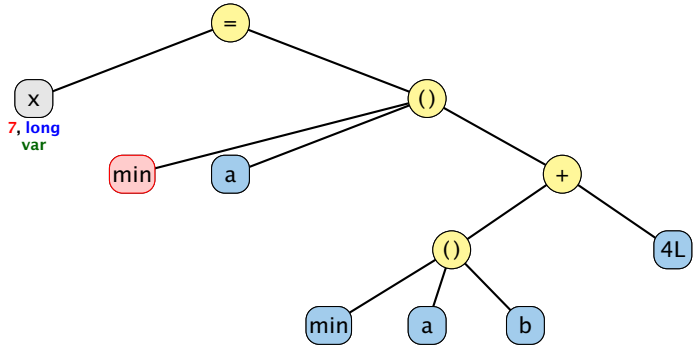
long x 7 int a 3 int b 5

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)




```
int min(int,int)
float min(float,float)
double min(double,double)
```

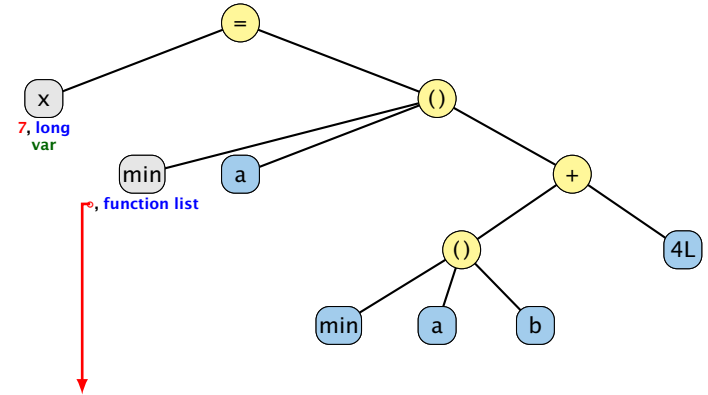
long x 7 int a 3 int b 5

5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

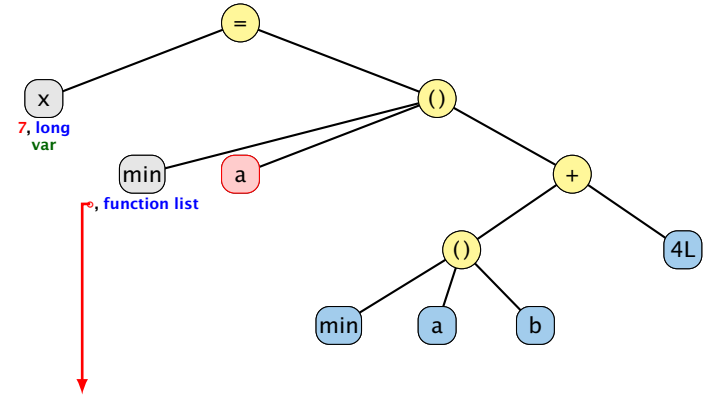


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

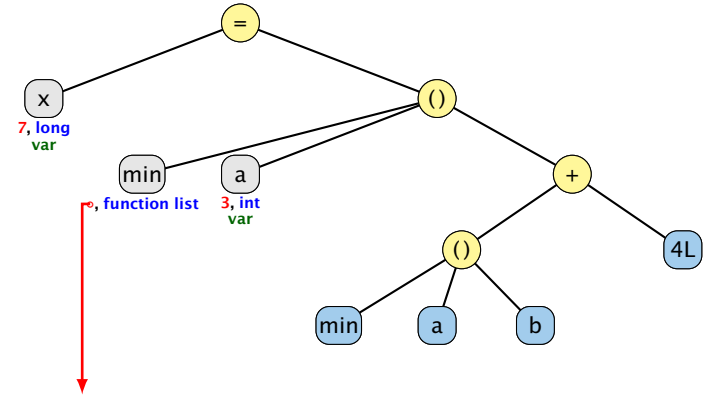


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

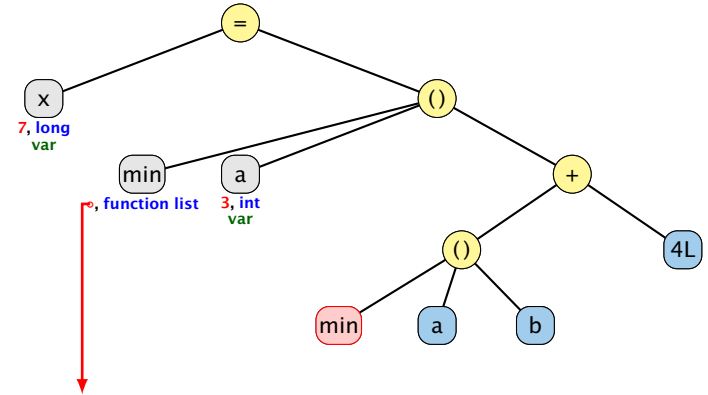


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

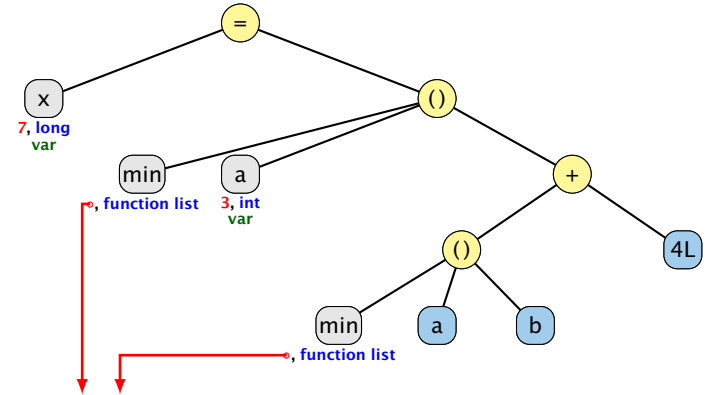


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

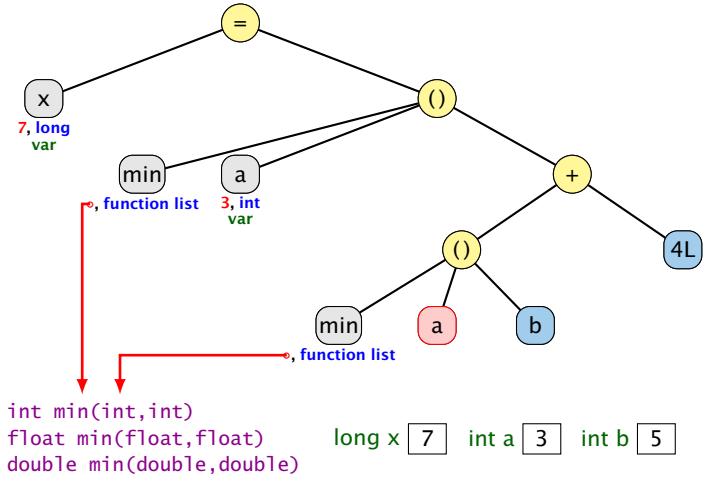


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

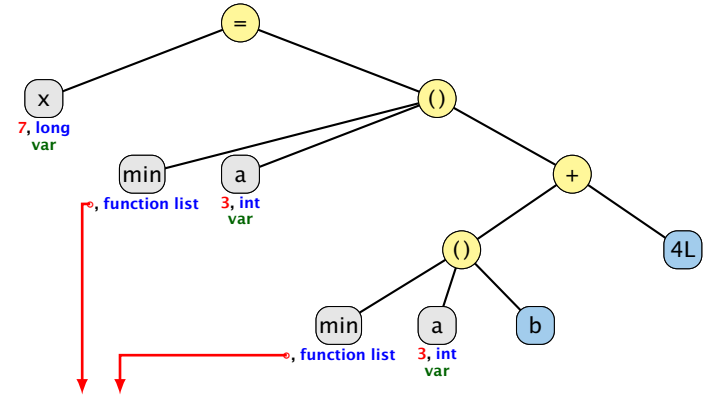


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

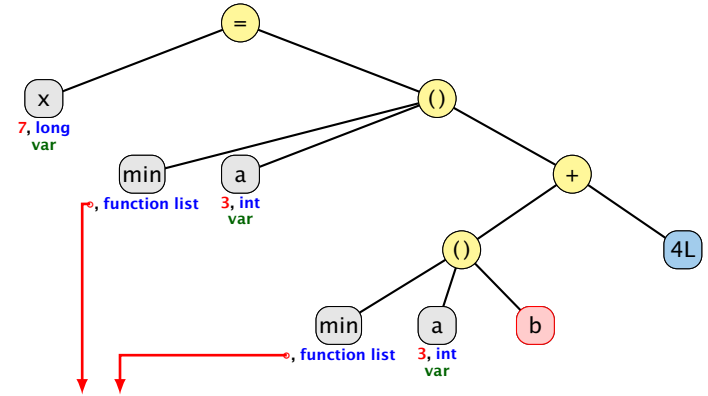


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

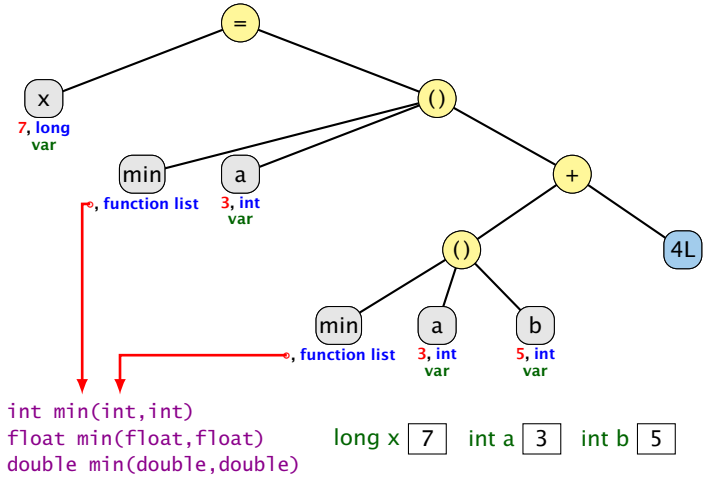


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

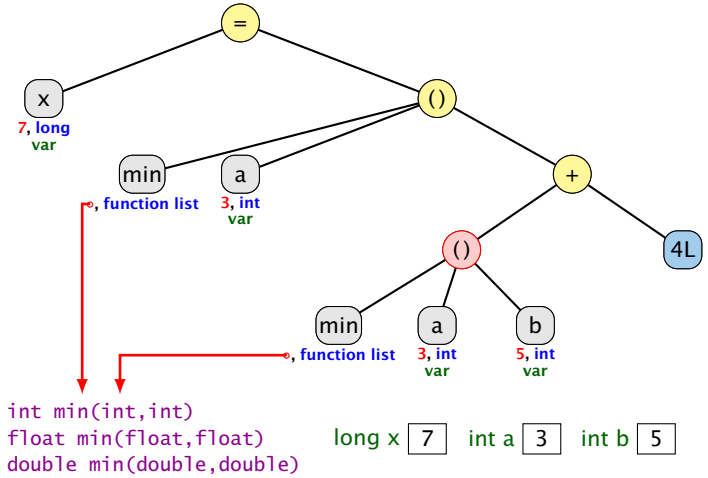


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

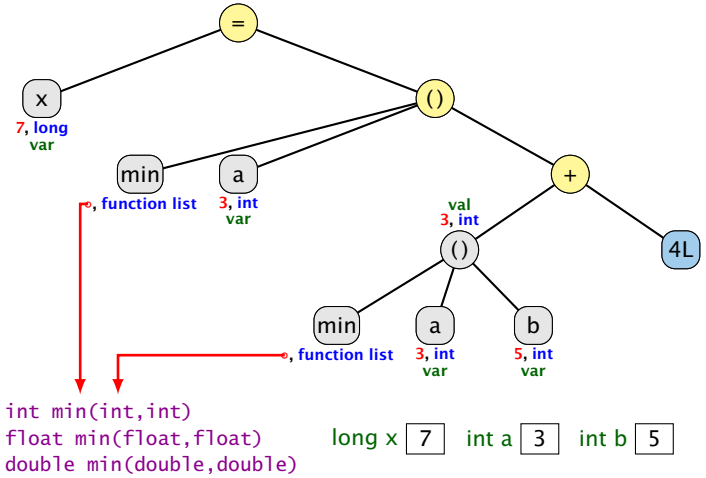


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

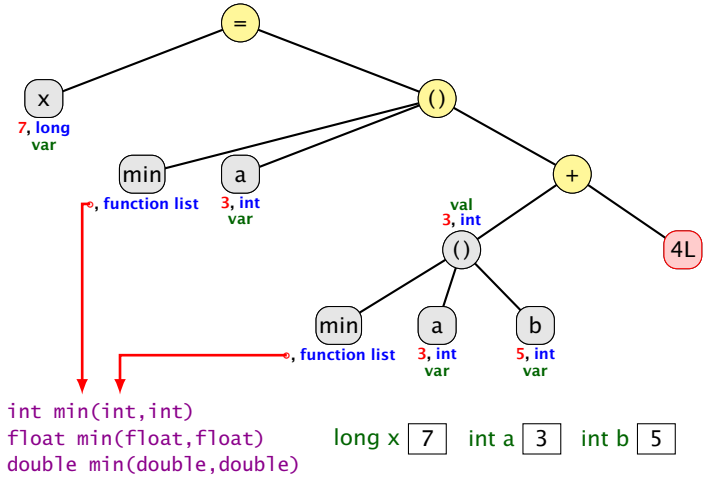


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

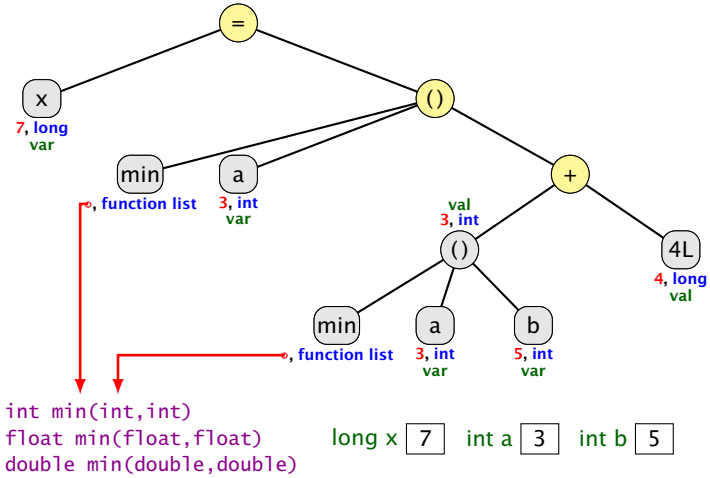


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

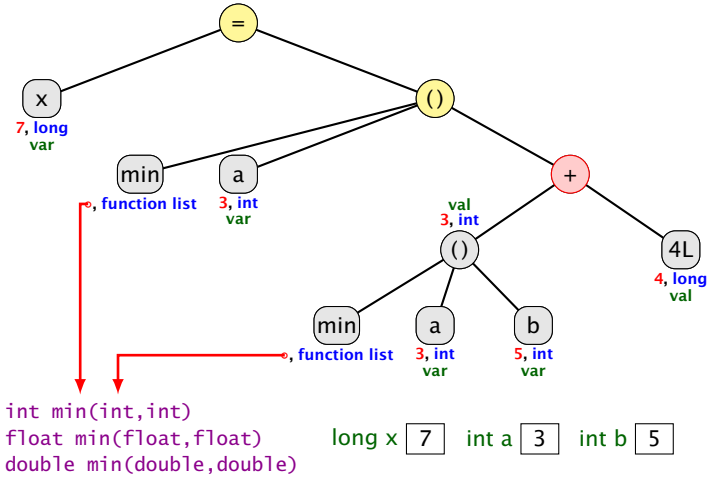


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

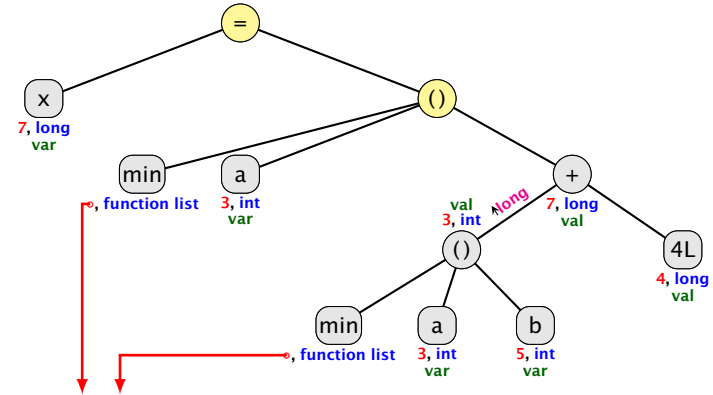


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

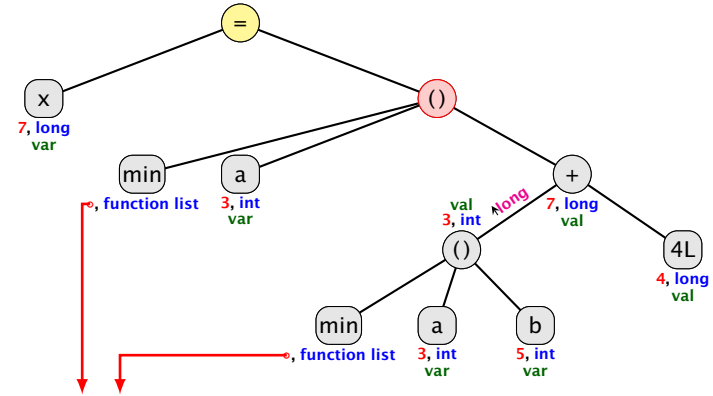


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

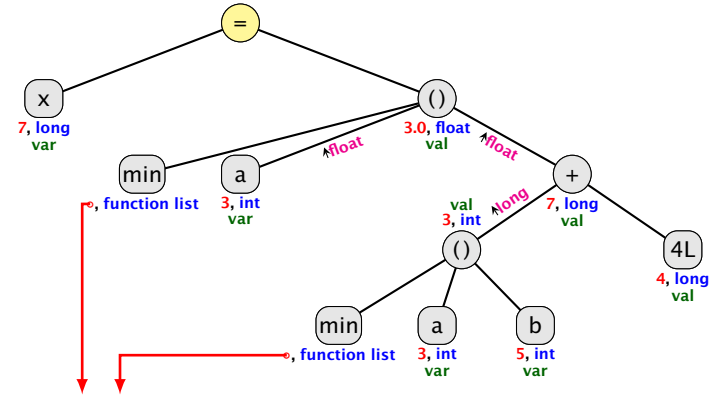


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

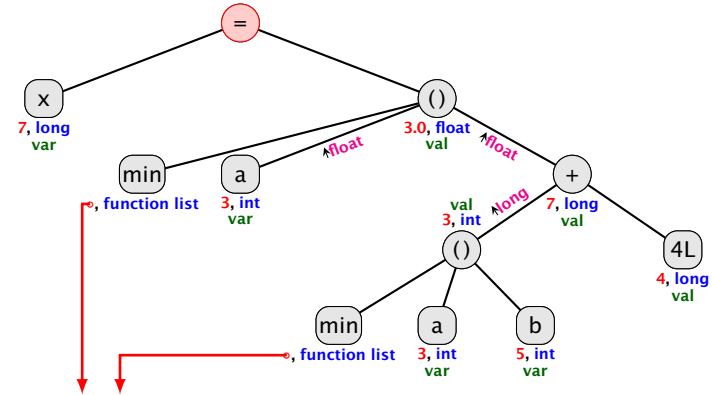


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)

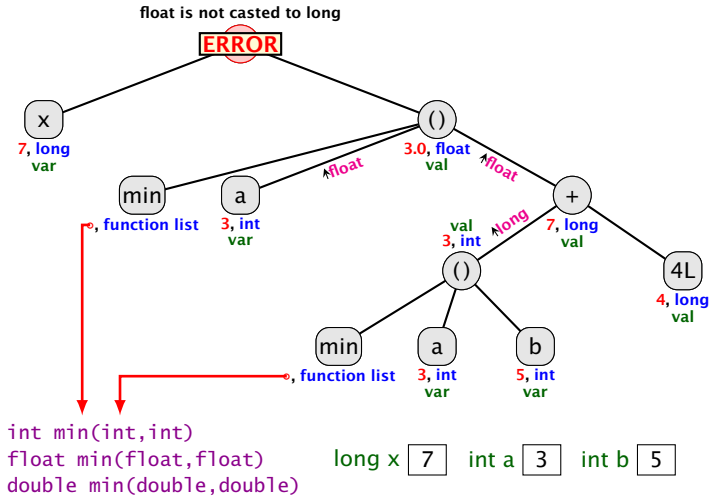


5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

Beispiel: x = min(a, min(a,b) + 4L)



5.3 Auswertung von Ausdrücken

Der Funktionsaufrufoperator:

symbol	*name*	*types*	*L/R*	*level*
( )	Funktionsaufruf	Funktionsname, *	links	1

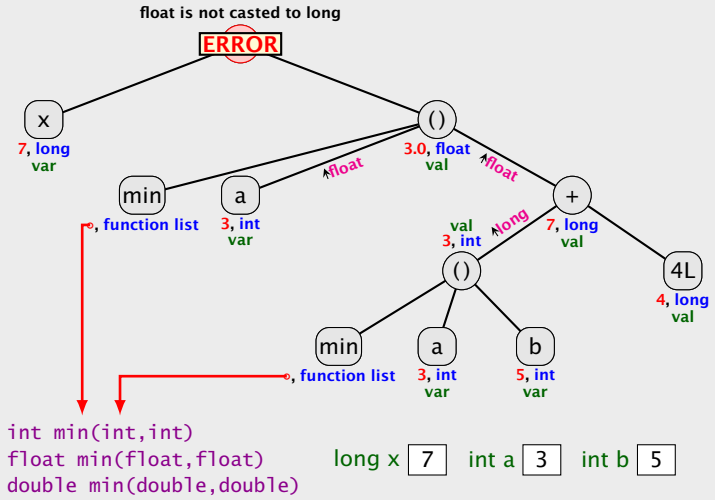
Impliziter Typecast – Strings

Spezialfall

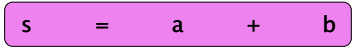
- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: x = min(a, min(a,b) + 4L)



Beispiel: s = a + b



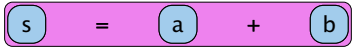
Impliziter Typecast – Strings

Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b



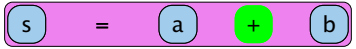
Impliziter Typecast – Strings

Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b



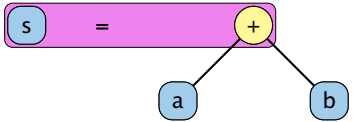
Impliziter Typecast – Strings

Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b



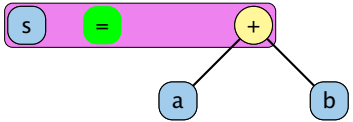
Impliziter Typecast – Strings

Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b



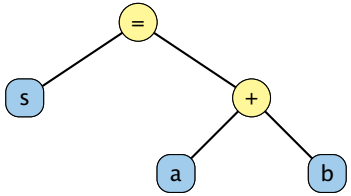
Impliziter Typecast – Strings

Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b



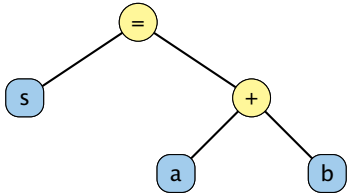
Impliziter Typecast – Strings

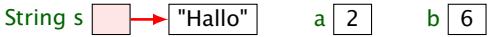
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





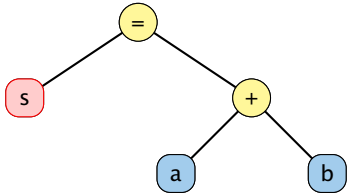
Impliziter Typecast – Strings

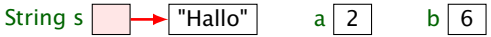
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





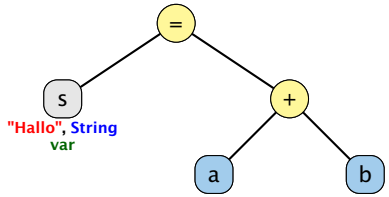
Impliziter Typecast – Strings

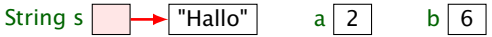
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





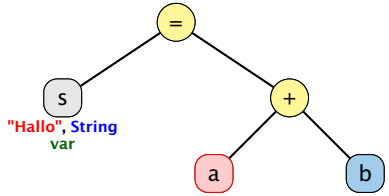
Impliziter Typecast – Strings

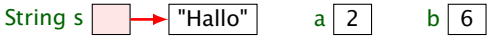
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





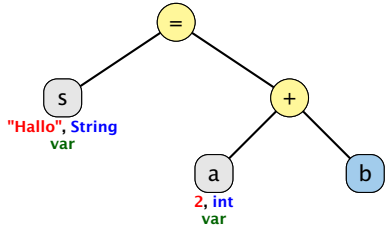
Impliziter Typecast – Strings

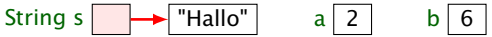
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





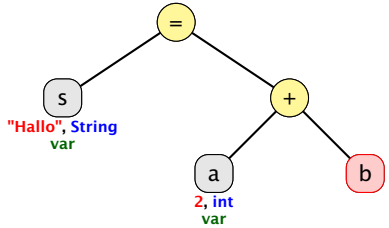
Impliziter Typecast – Strings

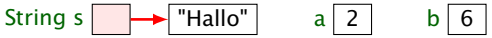
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





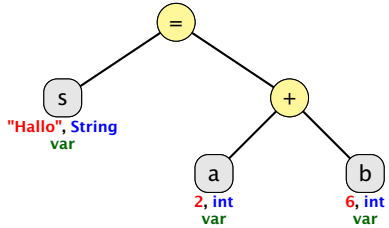
Impliziter Typecast – Strings

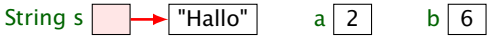
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





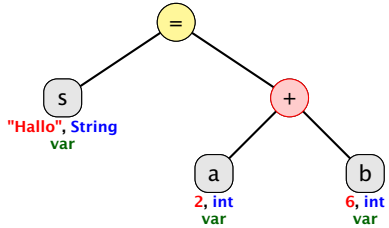
Impliziter Typecast – Strings

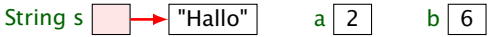
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





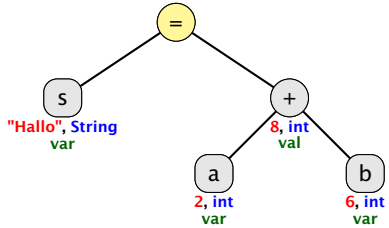
Impliziter Typecast – Strings

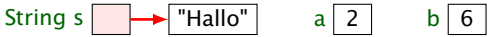
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





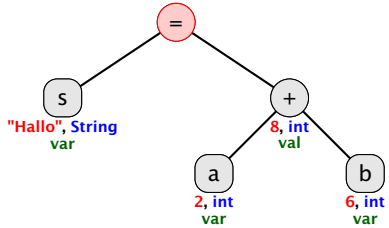
Impliziter Typecast – Strings

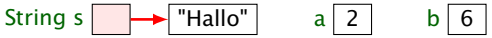
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





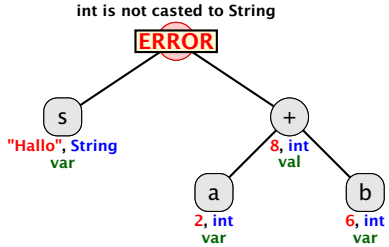
Impliziter Typecast – Strings

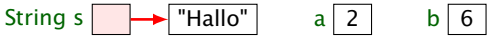
Spezialfall

- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = a + b





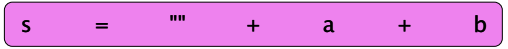
Impliziter Typecast – Strings

Spezialfall

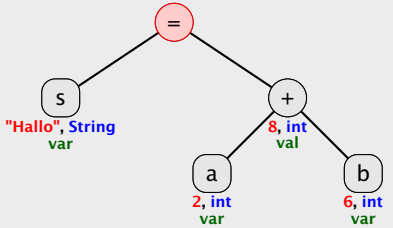
- Falls beim Operator + ein Typ vom Typ String ist, wird der andere auch in einen String umgewandelt.
  $\Longrightarrow$ Stringkonkatenation.
- Jeder Typ in Java besitzt eine Stringrepresentation.

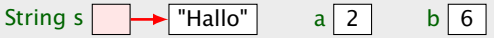
Funktioniert nicht bei selbstgeschriebenen Funktionen.

Beispiel: s = "" + a + b

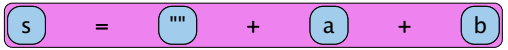


Beispiel: s = a + b

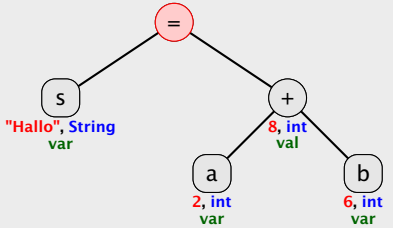


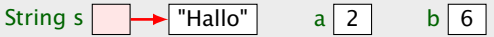


Beispiel: s = "" + a + b

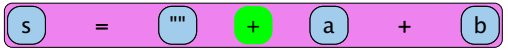


Beispiel: s = a + b

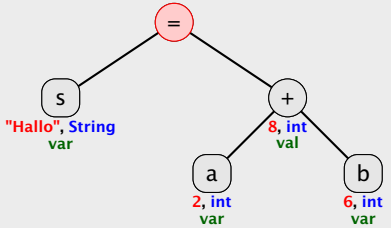




Beispiel: s = "" + a + b

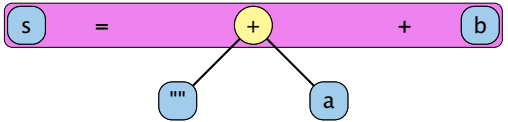


Beispiel: s = a + b

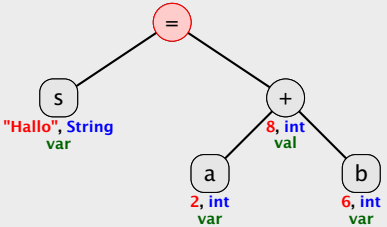


String s "Hallo" a 2 b 6

Beispiel: s = "" + a + b

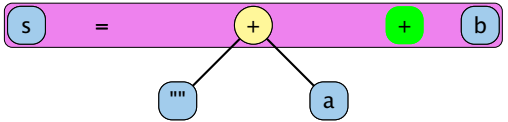


Beispiel: s = a + b

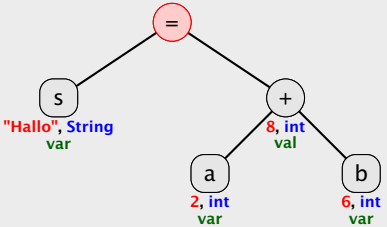


String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b

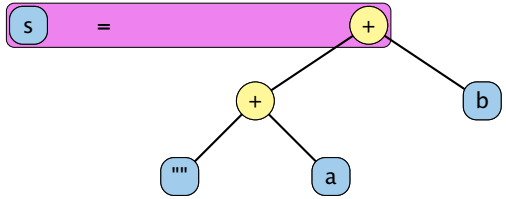


Beispiel: s = a + b

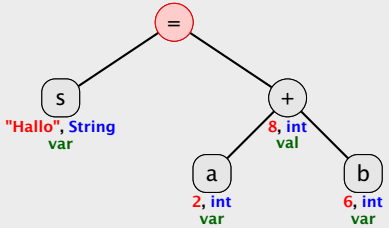


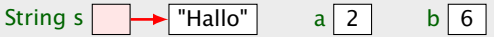
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b

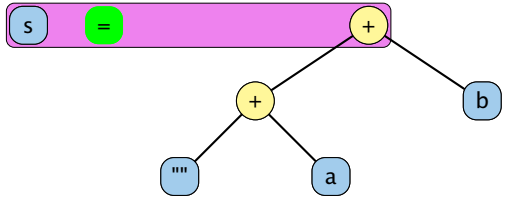


Beispiel: s = a + b

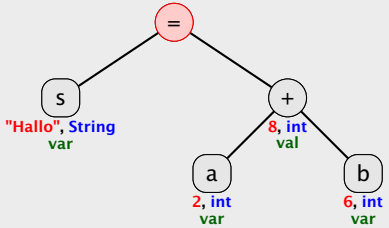




Beispiel: s = "" + a + b

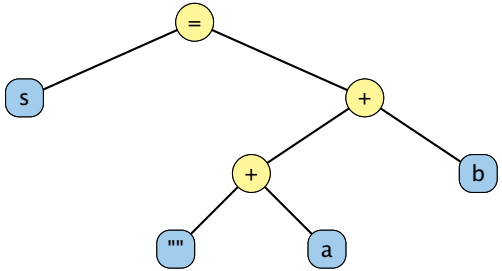


Beispiel: s = a + b



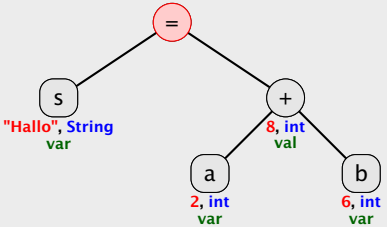
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



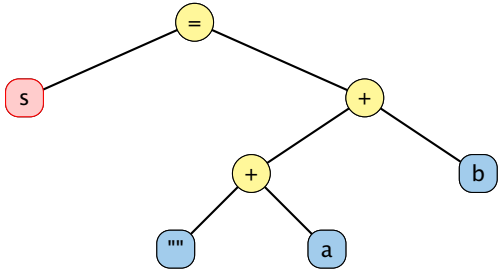
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



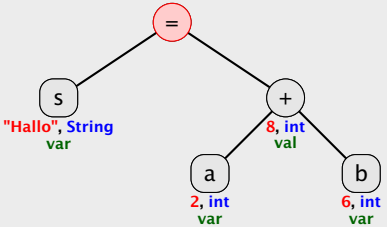
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



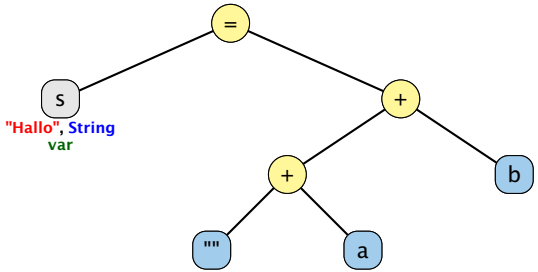
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



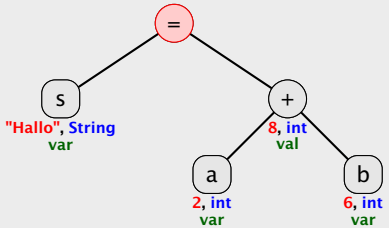
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



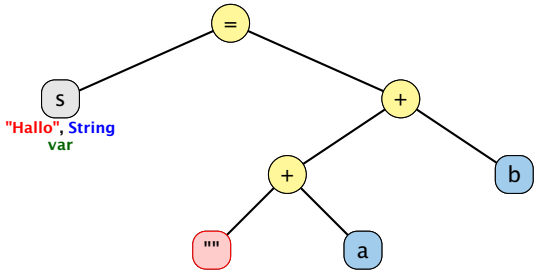
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



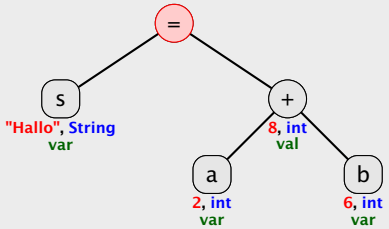
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



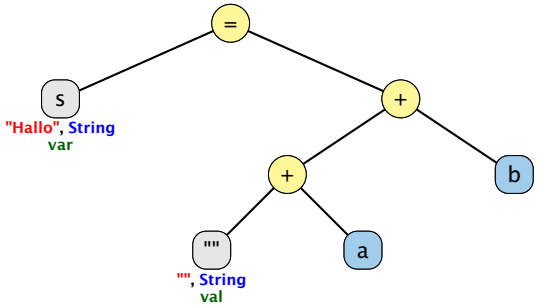
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



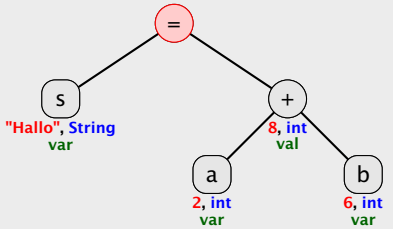
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



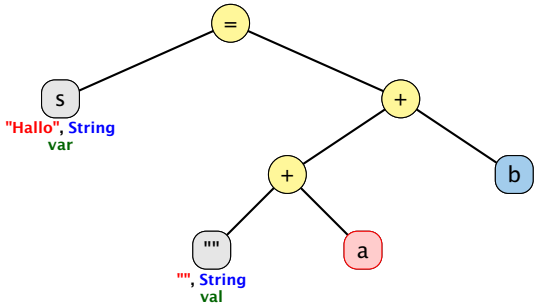
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



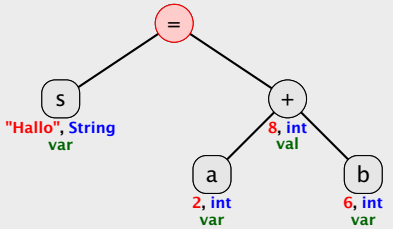
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



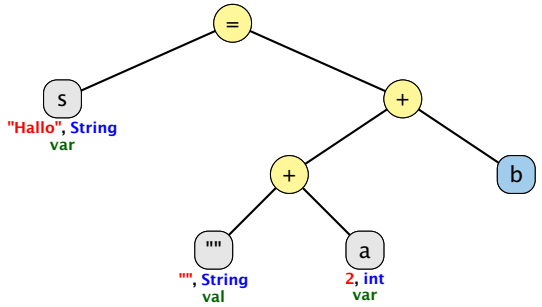
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



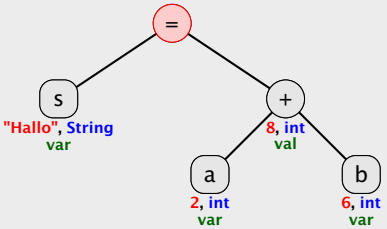
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



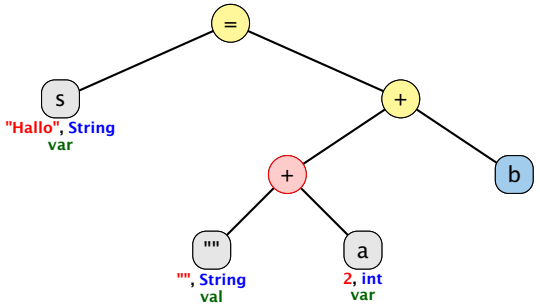
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



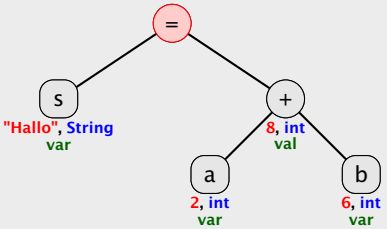
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



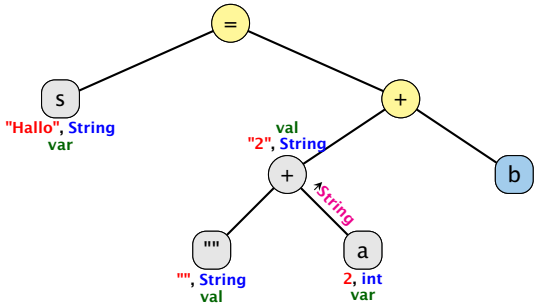
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



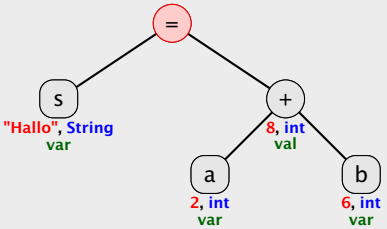
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



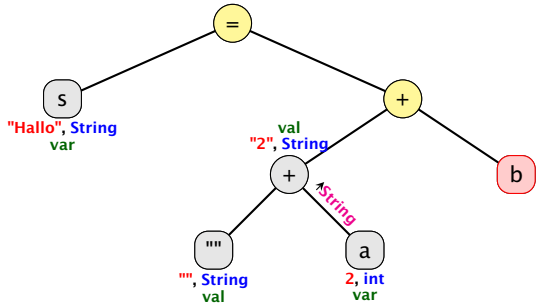
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



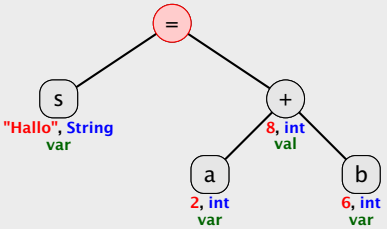
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



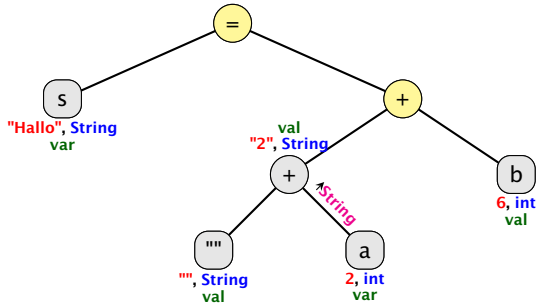
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



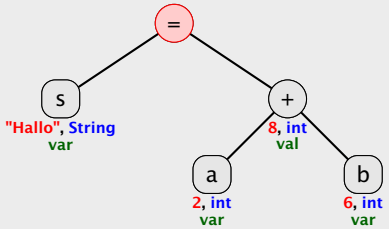
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



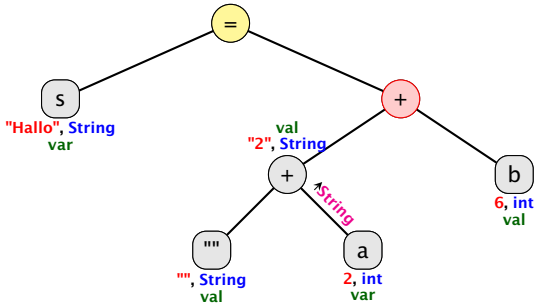
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



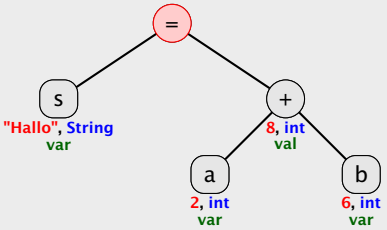
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



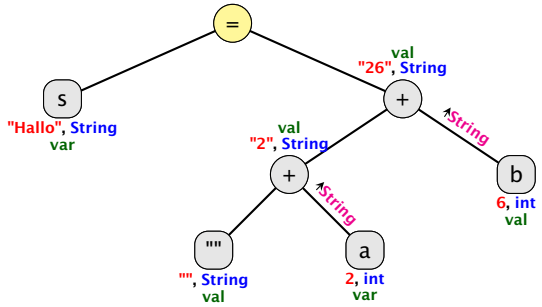
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



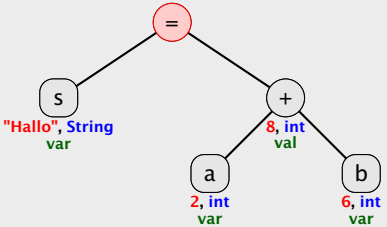
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



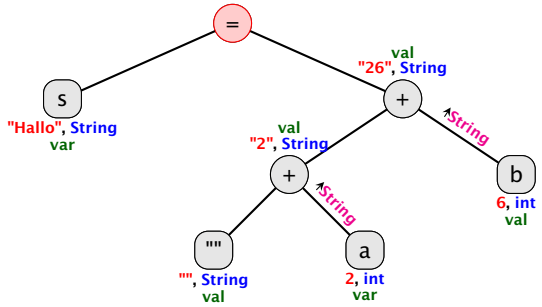
String s → "Hallo" a 2 b 6

Beispiel: s = a + b



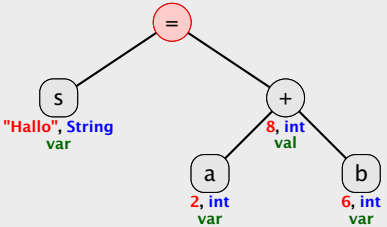
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b



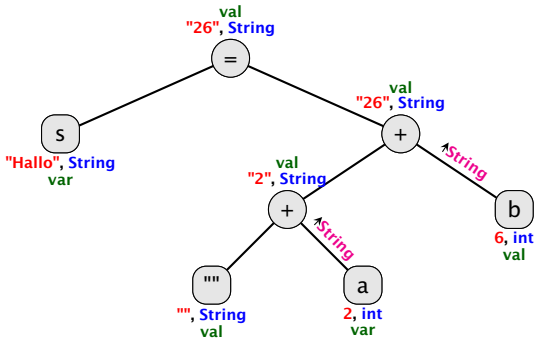
String s → "Hallo" a 2 b 6

Beispiel: s = a + b

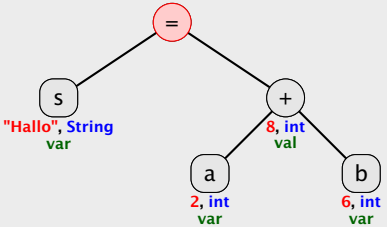


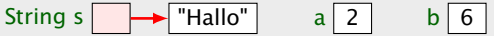
String s → "Hallo" a 2 b 6

Beispiel: s = "" + a + b

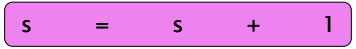


Beispiel: s = a + b

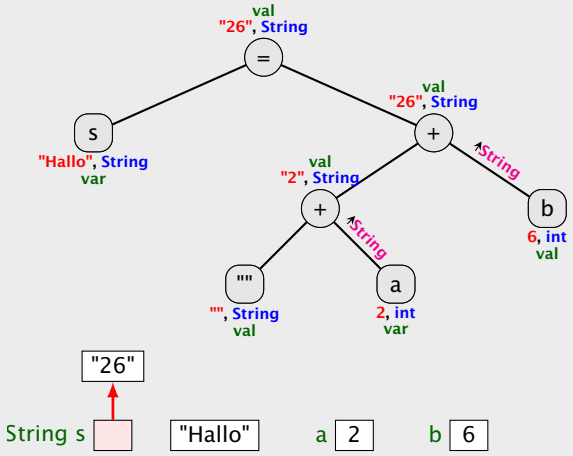




Beispiel: s = s + 1

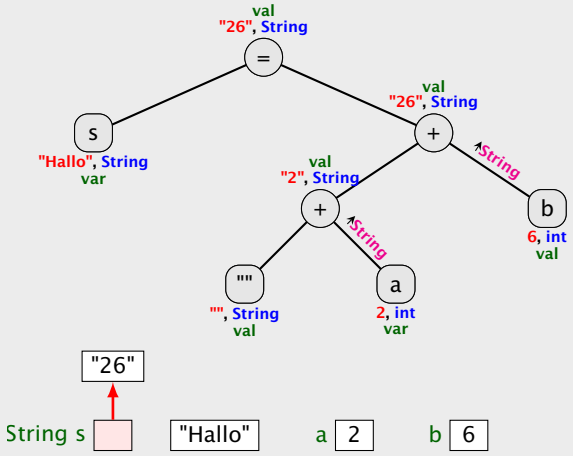


Beispiel: s = "" + a + b



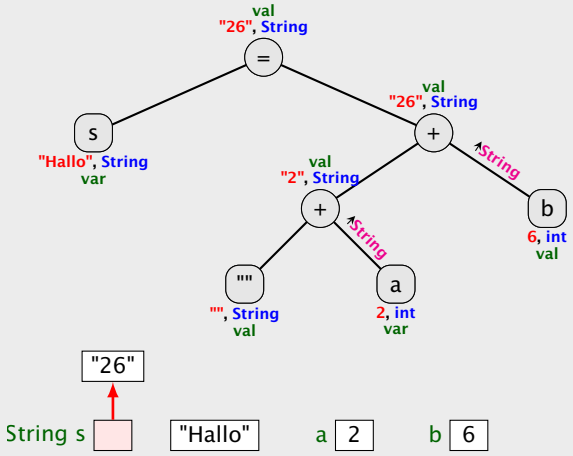
Beispiel: s = s + 1

Beispiel: s = "" + a + b

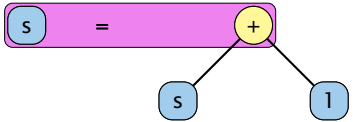


Beispiel: s = s + 1

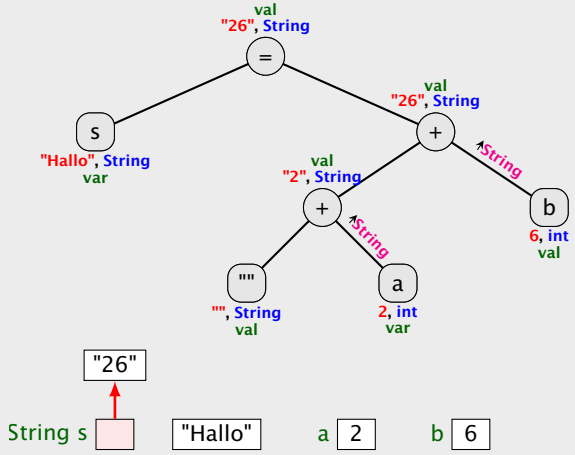
Beispiel: s = "" + a + b



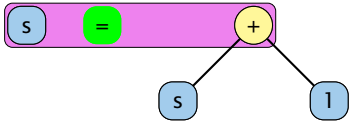
Beispiel: s = s + 1



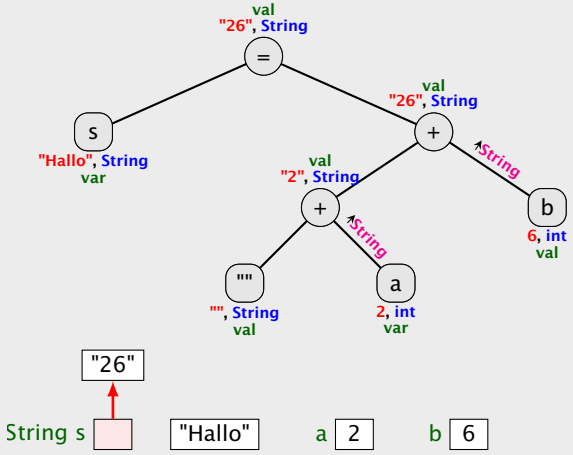
Beispiel: s = "" + a + b



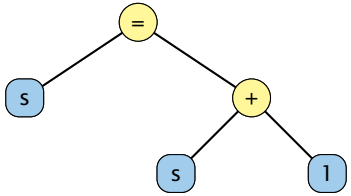
Beispiel: s = s + 1



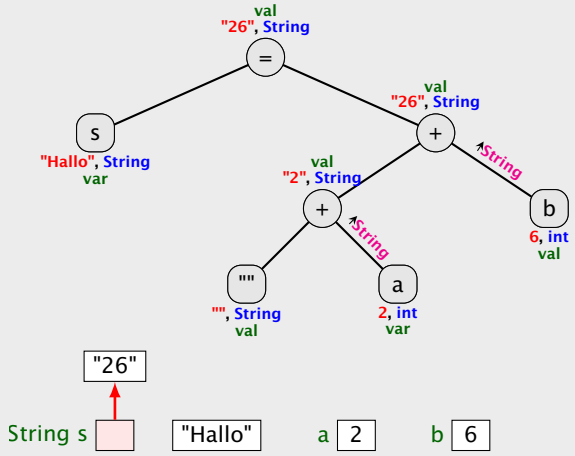
Beispiel: s = "" + a + b



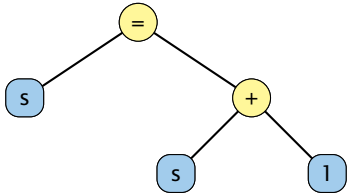
Beispiel: s = s + 1



Beispiel: s = "" + a + b

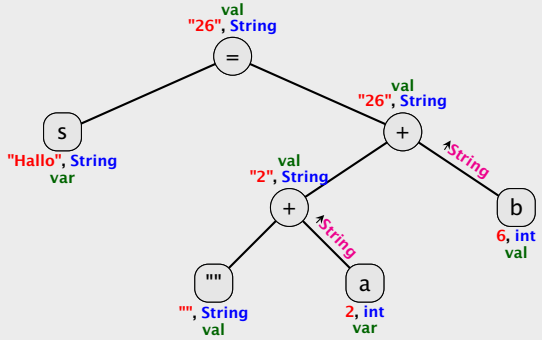


Beispiel: s = s + 1



short s 4

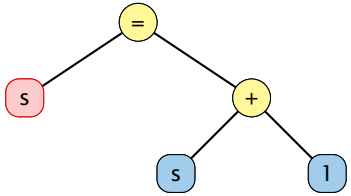
Beispiel: s = "" + a + b



"26"

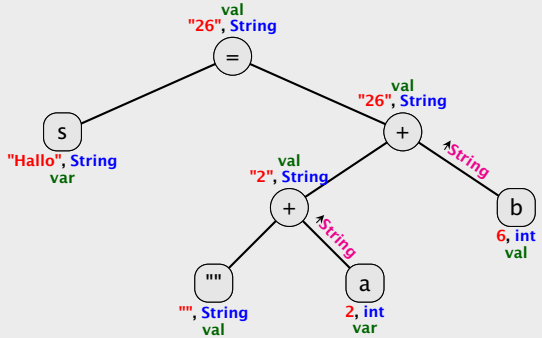
String s "Hallo" a 2 b 6

Beispiel: s = s + 1



short s 4

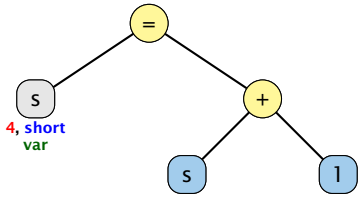
Beispiel: s = "" + a + b



"26"

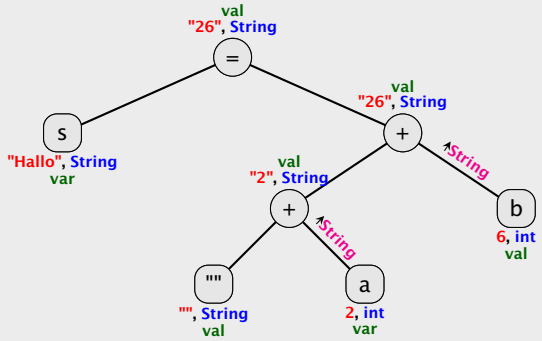
String s "Hallo" a 2 b 6

Beispiel: s = s + 1



short s 4

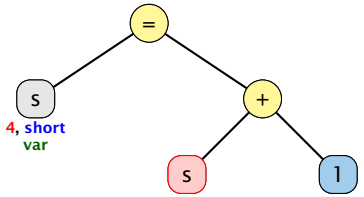
Beispiel: s = "" + a + b



"26"

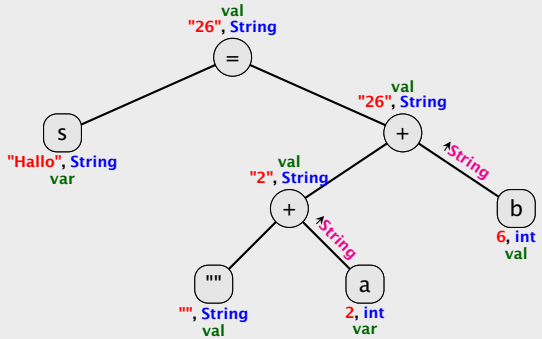
String s "Hallo" a 2 b 6

Beispiel: s = s + 1



short s 4

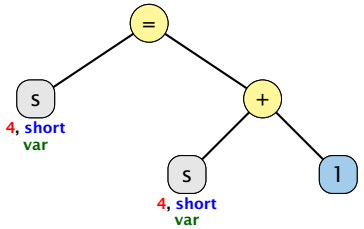
Beispiel: s = "" + a + b



"26"

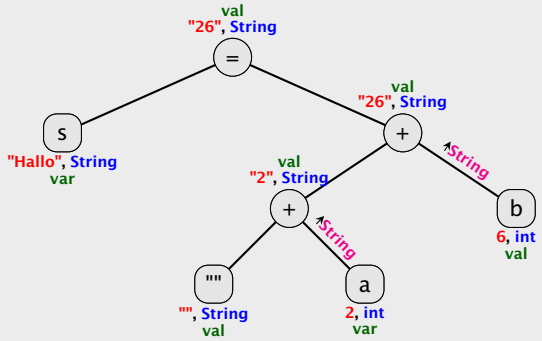
String s "Hallo" a 2 b 6

Beispiel: s = s + 1



short s | 4 |

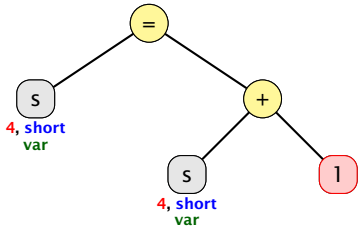
Beispiel: s = "" + a + b



"26"

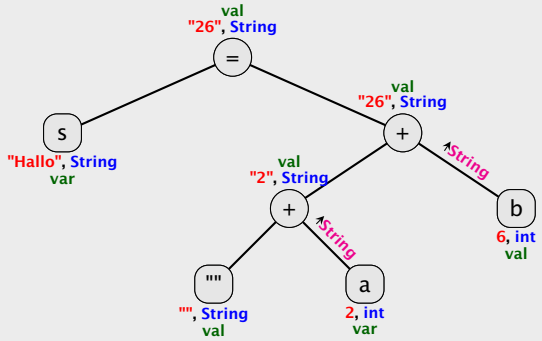
String s | "Hallo" | a | 2 | b | 6 |

Beispiel: s = s + 1



short s | 4 |

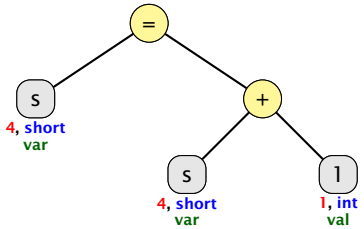
Beispiel: s = "" + a + b



"26"

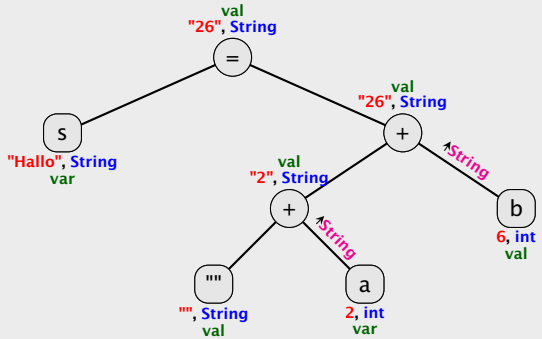
String s | "Hallo" | a | 2 | b | 6 |

Beispiel: s = s + 1



short s 4

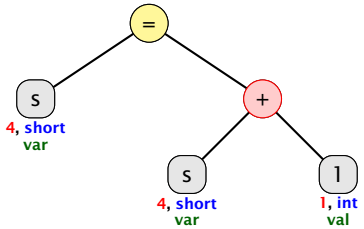
Beispiel: s = "" + a + b



"26"

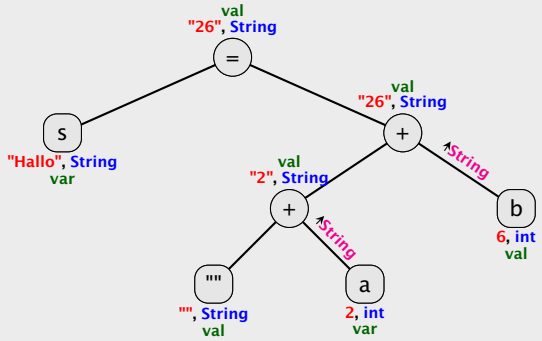
String s "Hallo" a 2 b 6

Beispiel: s = s + 1



short s | 4 |

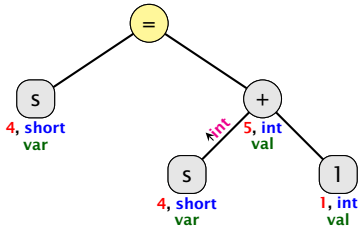
Beispiel: s = "" + a + b



"26"

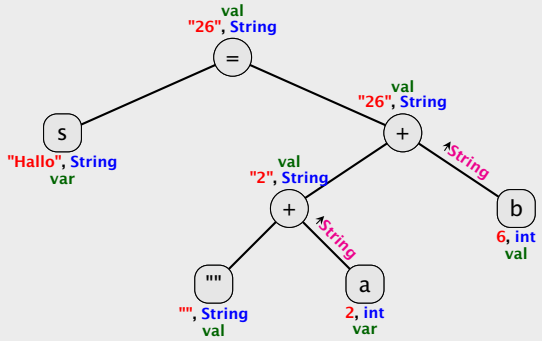
String s | "Hallo" | a | 2 | b | 6 |

Beispiel: s = s + 1



short s 4

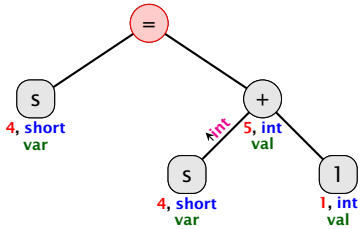
Beispiel: s = "" + a + b



"26"

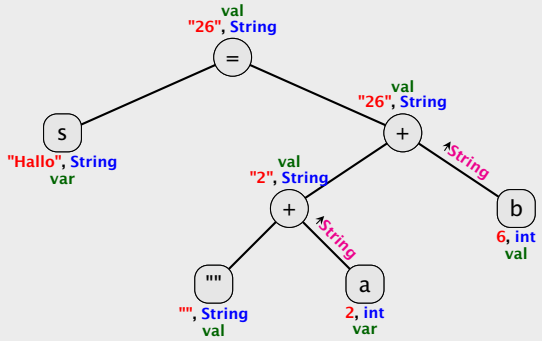
String s "Hallo" a 2 b 6

Beispiel: s = s + 1



short s | 4 |

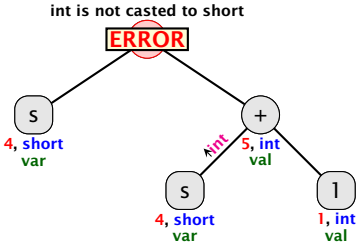
Beispiel: s = "" + a + b



"26"

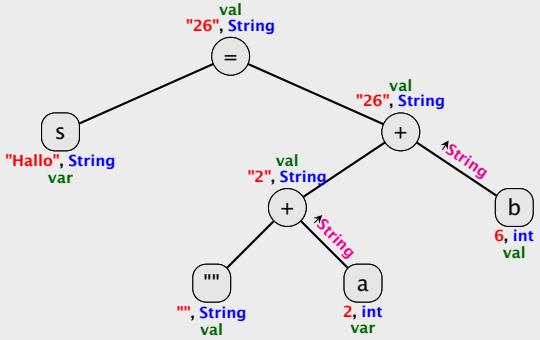
String s | "Hallo" | a | 2 | b | 6 |

Beispiel: s = s + 1



short s | 4 |

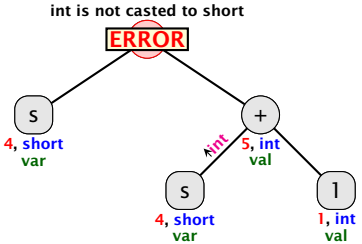
Beispiel: s = "" + a + b



"26"

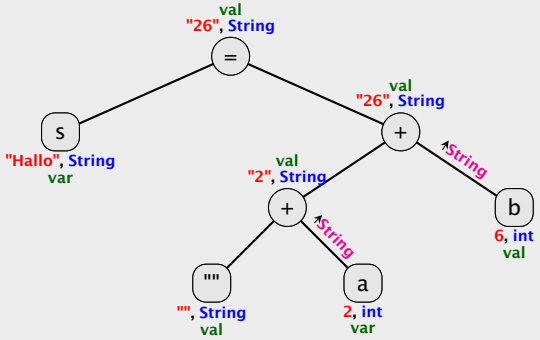
String s | "Hallo" | a | 2 | b | 6 |

Beispiel: s = s + 1



short s | 4 |

Beispiel: s = "" + a + b

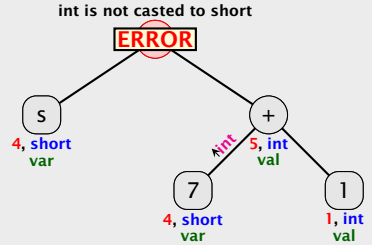


"26"

String s | "Hallo" | a | 2 | b | 6 |

Beispiel: s = 7 + 1

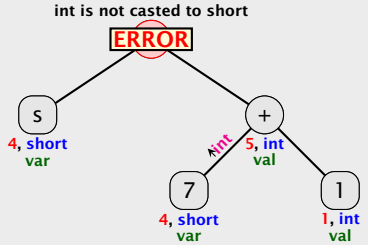
Beispiel: s = s + 1



short s 4

Beispiel: s = 7 + 1

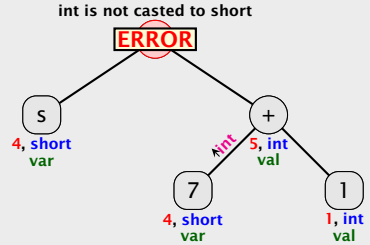
Beispiel: s = s + 1



short s 4

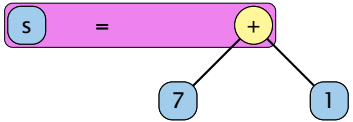
Beispiel: s = 7 + 1

Beispiel: s = s + 1

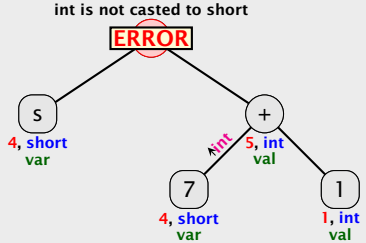


short s 4

Beispiel: s = 7 + 1

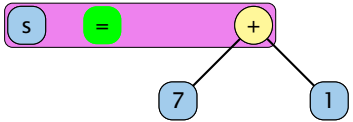


Beispiel: s = s + 1

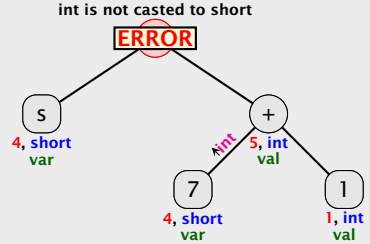


short s 4

Beispiel: s = 7 + 1

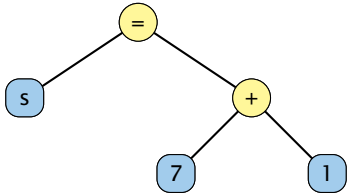


Beispiel: s = s + 1

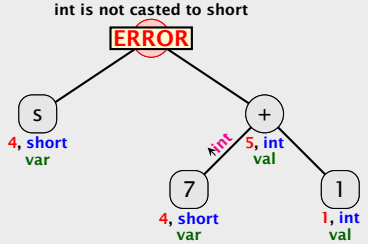


short s 4

Beispiel: s = 7 + 1

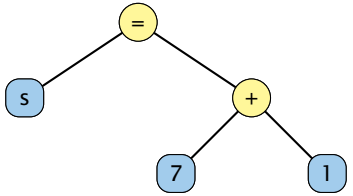


Beispiel: s = s + 1



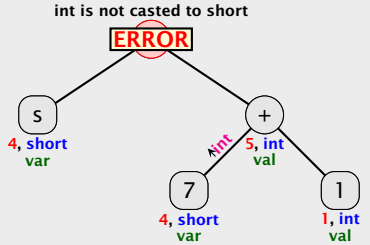
short s 4

Beispiel: s = 7 + 1



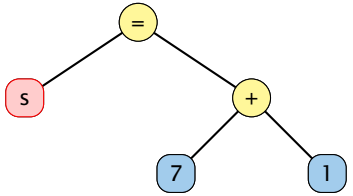
short s 4

Beispiel: s = s + 1



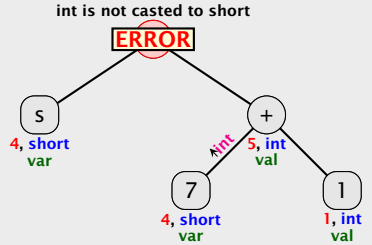
short s 4

Beispiel: s = 7 + 1



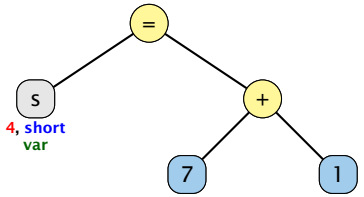
short s 4

Beispiel: s = s + 1



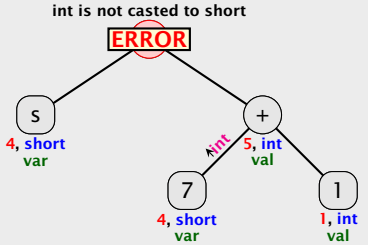
short s 4

Beispiel: s = 7 + 1



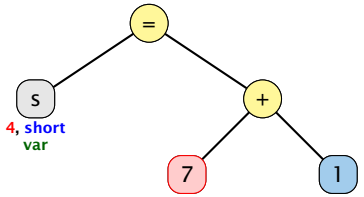
short s 4

Beispiel: s = s + 1



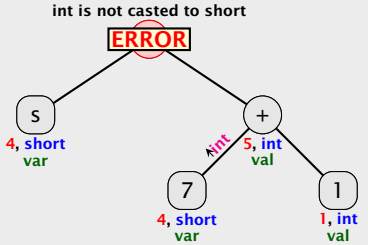
short s 4

Beispiel: s = 7 + 1



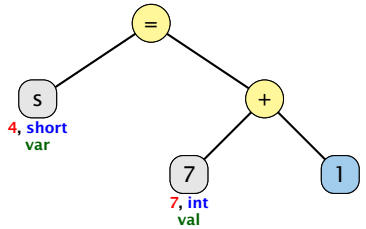
short s 4

Beispiel: s = s + 1



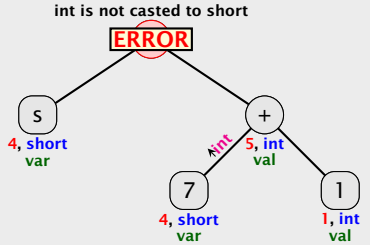
short s 4

Beispiel: s = 7 + 1



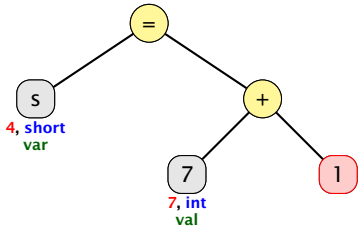
short s 4

Beispiel: s = s + 1



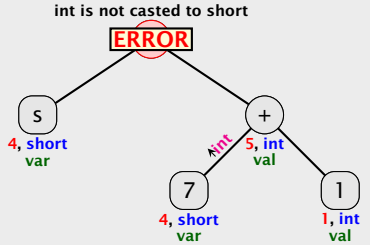
short s 4

Beispiel: s = 7 + 1



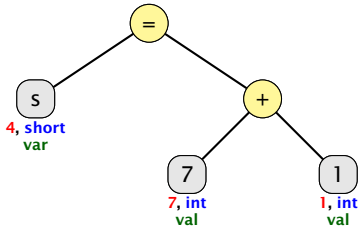
short s 4

Beispiel: s = s + 1



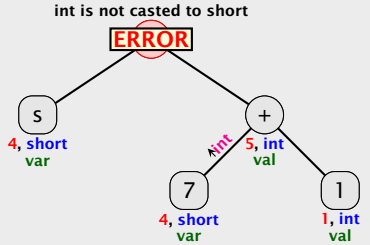
short s 4

Beispiel: s = 7 + 1



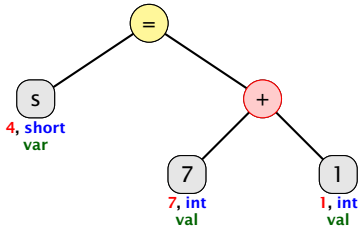
short s 4

Beispiel: s = s + 1



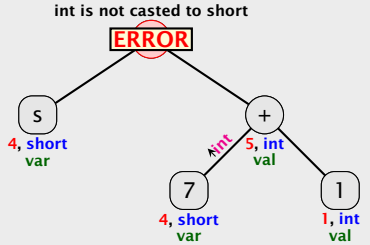
short s 4

Beispiel: s = 7 + 1



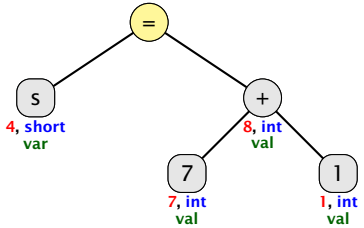
short s 4

Beispiel: s = s + 1



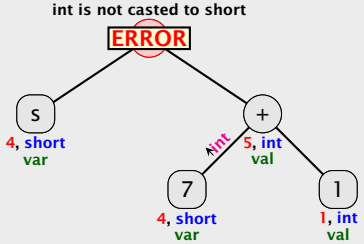
short s 4

Beispiel: s = 7 + 1



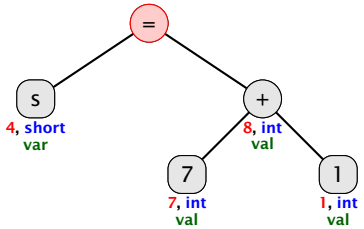
short s 4

Beispiel: s = s + 1



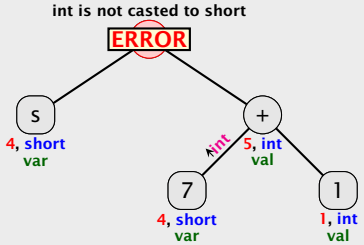
short s 4

Beispiel: s = 7 + 1



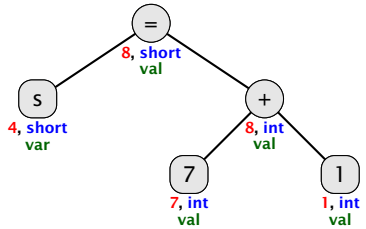
short s 4

Beispiel: s = s + 1



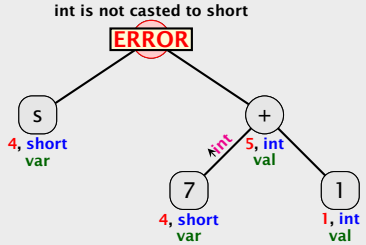
short s 4

Beispiel: s = 7 + 1



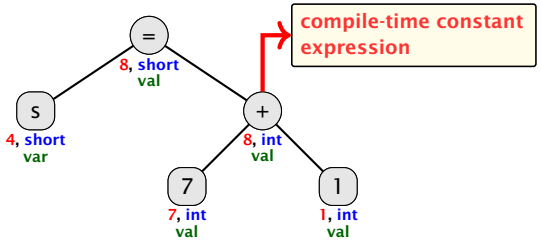
short s 8

Beispiel: s = s + 1



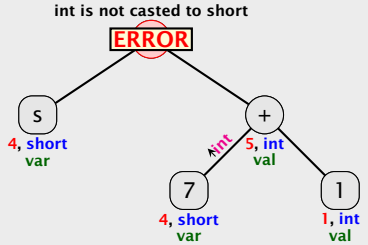
short s 4

Beispiel: s = 7 + 1



short s 8

Beispiel: s = s + 1



short s 4

Expliziter Typecast

symbol	*name*	*type*	*L/R*	*level*
(type)	typecast	zahl, char	rechts	3

Beispiele mit Datenverlust

- `short s = (short) 23343445;`

  Die obersten bits werden einfach weggeworfen...

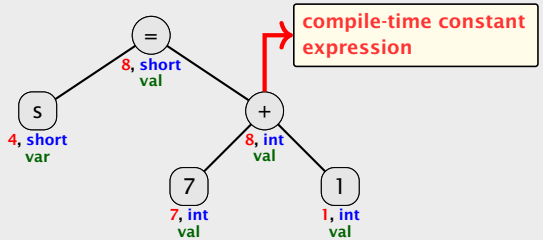
- ```
  double d = 1.5;
  short  s = (short) d;
  ```

  d hat danach den Wert 1.

...ohne Datenverlust:

- ```
  int   x = 5;
  short s = (short) x;
  ```

Beispiel: s = 7 + 1



short s 8

5.4 Arrays

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!

Feld:	17	3	-2	9	0	1
Index:	0	1	2	3	4	5

Expliziter Typecast

symbol	*name*	*type*	*L/R*	*level*
(type)	typecast	zahl, char	rechts	3

Beispiele mit Datenverlust

- ```
  short s = (short) 23343445;
  ```
  Die obersten bits werden einfach weggeworfen...
- ```
  double d = 1.5;
  short  s = (short) d;
  ```
  d hat danach den Wert 1.

...ohne Datenverlust:

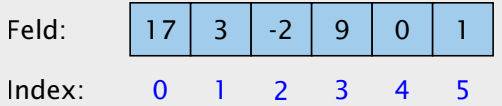
- ```
  int   x = 5;
  short s = (short) x;
  ```
  ```

5.4 Arrays

Oft müssen viele Werte gleichen Typs gespeichert werden.

Idee:

- Lege sie konsekutiv ab!
- Greife auf einzelne Werte über ihren Index zu!



Beispiel

```
1 int[] a; // Deklaration
2 int n = read();
3
4 a = new int[n]; // Anlegen des Felds
5 int i = 0;
6 while (i < n) {
7     a[i] = read();
8     i = i+1;
9 }
```

Einlesen eines Feldes

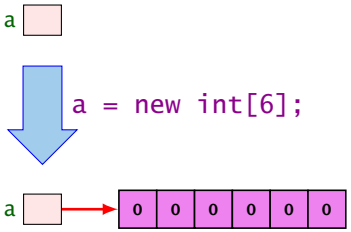
Beispiel

```
1 int[] a; // Deklaration
2 int n = read();
3
4 a = new int[n]; // Anlegen des Felds
5 int i = 0;
6 while (i < n) {
7     a[i] = read();
8     i = i+1;
9 }
```

Einlesen eines Feldes

Beispiel

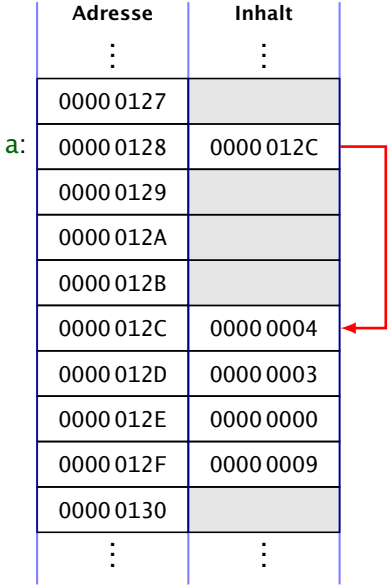
- `type[] name;` deklariert eine Variable für ein Feld (array), dessen Elemente vom Typ `type` sind.
- Alternative Schreibweise:
  `type name[];`
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen Verweis darauf zurück:



Was ist eine Referenz?

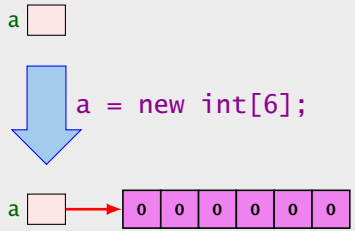
Eine Referenzvariable speichert eine Adresse; an dieser Adresse liegt der eigentliche Inhalt der Variablen.

Wir können die Referenz nicht direkt manipulieren (nur über den new-Operator, oder indem wir eine andere Referenz zuweisen).



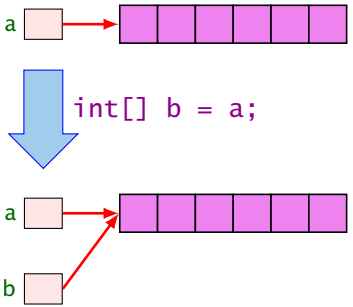
Beispiel

- `type[] name;` deklariert eine Variable für ein Feld (array), dessen Elemente vom Typ `type` sind.
- Alternative Schreibweise: `type name[];`
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen Verweis darauf zurück:



5.4 Arrays

- Der Wert einer Feld-Variable ist also ein Verweis!!!
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



- **Alle nichtprimitive Datentypen sind Referenzentypen, d.h., die zugehörige Variable speichert einen Verweis!!!**

Was ist eine Referenz?

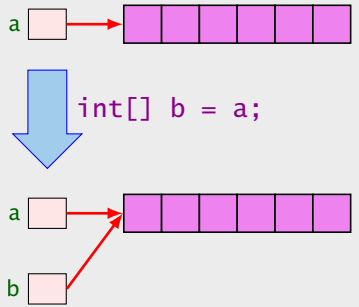
Eine Referenzvariable speichert eine Adresse; an dieser Adresse liegt der eigentliche Inhalt der Variablen.

Wir können die Referenz nicht direkt manipulieren (nur über den new-Operator, oder indem wir eine andere Referenz zuweisen).

	Adresse	Inhalt
	⋮	⋮
	0000 0127	
a:	0000 0128	0000 012C
	0000 0129	
	0000 012A	
	0000 012B	
	0000 012C	0000 0004
	0000 012D	0000 0003
	0000 012E	0000 0000
	0000 012F	0000 0009
	0000 0130	
	⋮	⋮

5.4 Arrays

- Der Wert einer Feld-Variable ist also ein Verweis!!!
- `int[] b = a;` kopiert den Verweis der Variablen `a` in die Variable `b`:



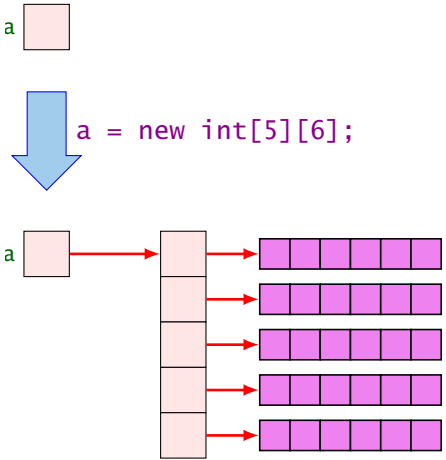
- **Alle nichtprimitive Datentypen sind Referenzentypen, d.h., die zugehörige Variable speichert einen Verweis!!!**

5.4 Arrays

- Die Elemente eines Feldes sind von 0 an durchnummeriert.
- Die Anzahl der Elemente des Feldes `name` ist `name.length`.
- Auf das i-te Element greift man mit `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0,\ldots,$`name.length-1`$\}$ liegt.
- Liegt der Index außerhalb des Intervalls, wird eine `ArrayIndexOutOfBoundException` ausgelöst (↑Exceptions).

Mehrdimensionale Felder

- Java unterstützt direkt nur eindimensionale Felder.
- ein zweidimensionales Feld ist ein Feld von Feldern...



5.4 Arrays

- Die Elemente eines Feldes sind von 0 an durchnummeriert.
- Die Anzahl der Elemente des Feldes `name` ist `name.length`.
- Auf das i-te Element greift man mit `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0,\ldots,$`name.length-1`$\}$ liegt.
- Liegt der Index außerhalb des Intervalls, wird eine `ArrayIndexOutOfBoundException` ausgelöst (↑Exceptions).

Der new-Operator

symbol	*name*	*types*	*L/R*	*level*
new	new	Typ, Konstruktor	links	1

Erzeugt ein Objekt/Array und liefert eine Referenz darauf zurück.

1. Version: Erzeugung eines Arrays (Typ ist Arraytyp)

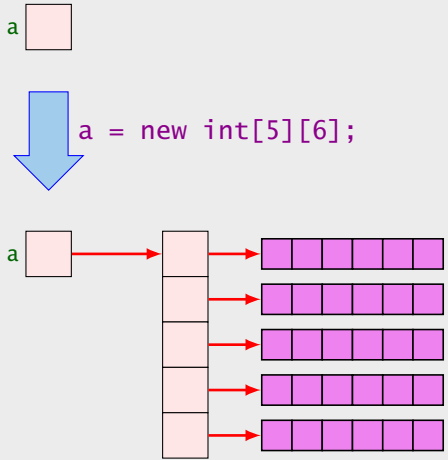
- `new int[3][7];` oder auch
- `new int[3][];` (ein Array, das 3 Verweise auf `int` enthält)
- `new String[10];`
- `new int[]{1,2,3,};` (ein Array mit den `int`s 1, 2, 3)

2. Version: Erzeugung eines Objekts durch Aufruf eines Konstruktors

- `String s = new String("Hello World!");`

Mehrdimensionale Felder

- Java unterstützt direkt nur eindimensionale Felder.
- ein zweidimensionales Feld ist ein Feld von Feldern...



Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Der new-Operator

symbol	*name*	*types*	*L/R*	*level*
new	new	Typ, Konstruktor	links	1

Erzeugt ein Objekt/Array und liefert eine Referenz darauf zurück.

1. Version: Erzeugung eines Arrays (Typ ist Arraytyp)
   - `new int[3][7];` oder auch
   - `new int[3][];` (ein Array, das 3 Verweise auf `int` enthält)
   - `new String[10];`
   - `new int[]{1,2,3,};` (ein Array mit den `int`s `1`, `2`, `3`)

2. Version: Erzeugung eines Objekts durch Aufruf eines Konstruktors
   - `String s = new String("Hello World!");`

Beispiel: x = a[3][2]

Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

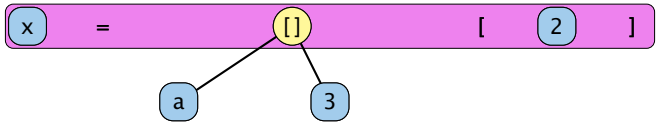
Beispiel: x = a[3][2]

Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

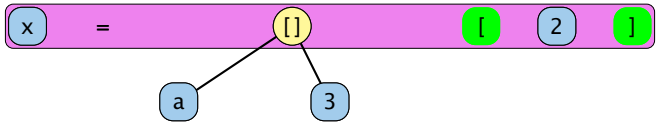


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

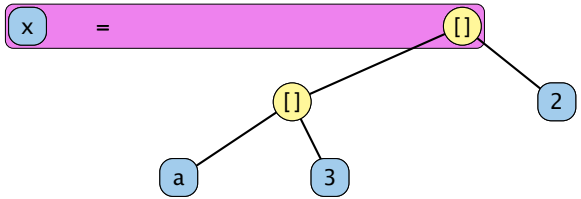


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

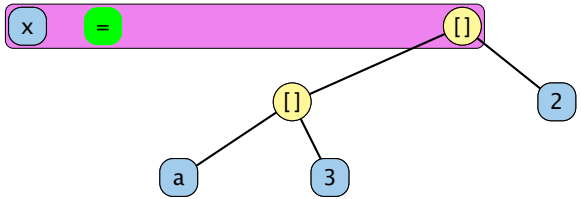


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

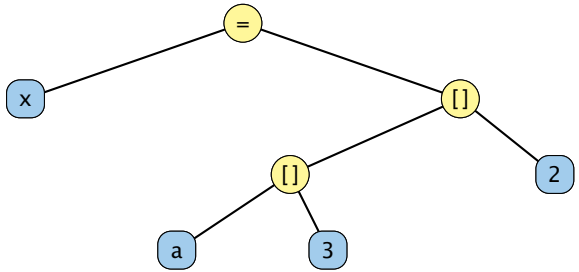


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

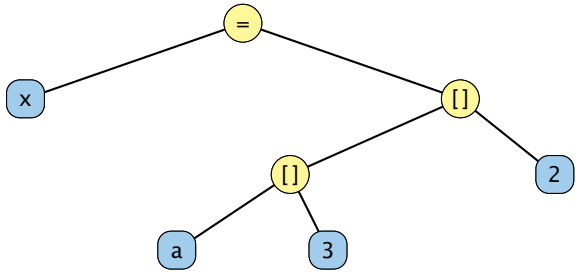


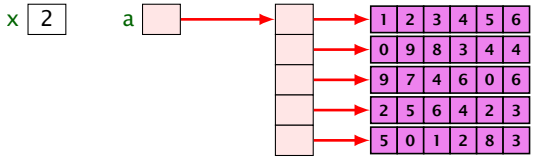
Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



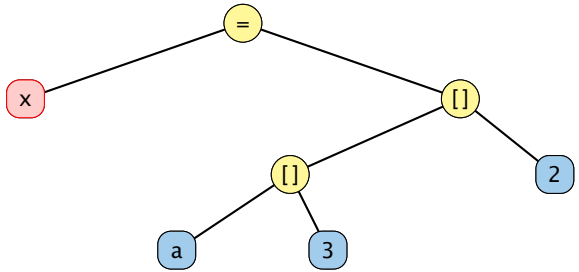


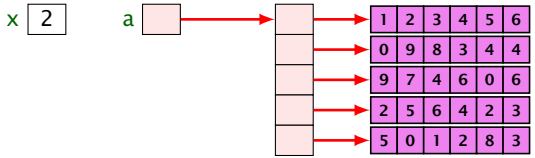
Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



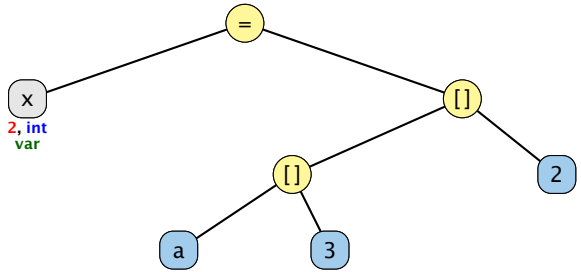


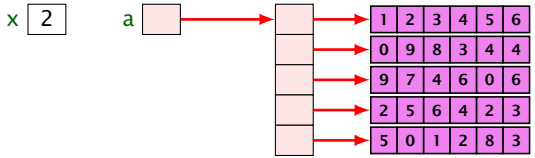
Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



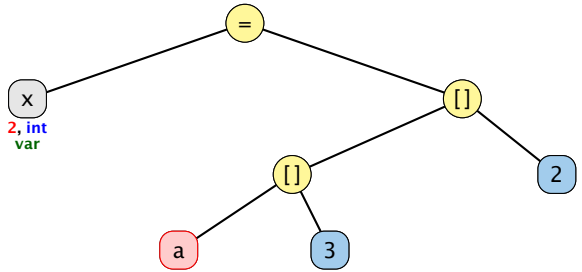


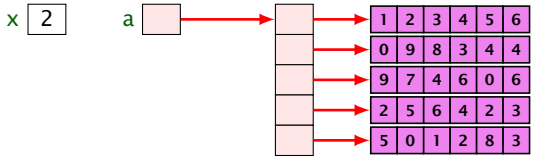
Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



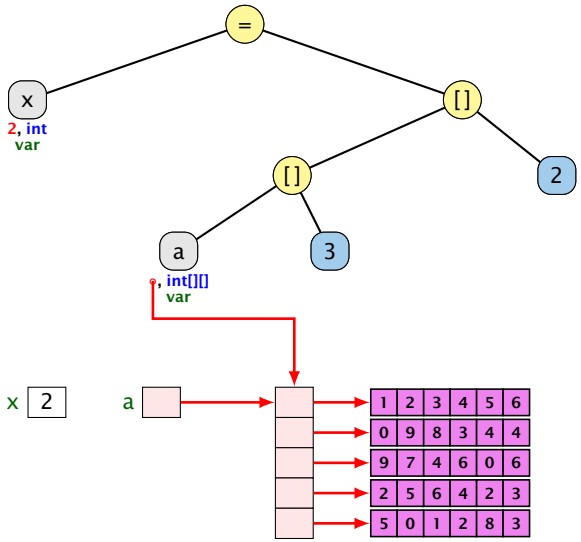


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

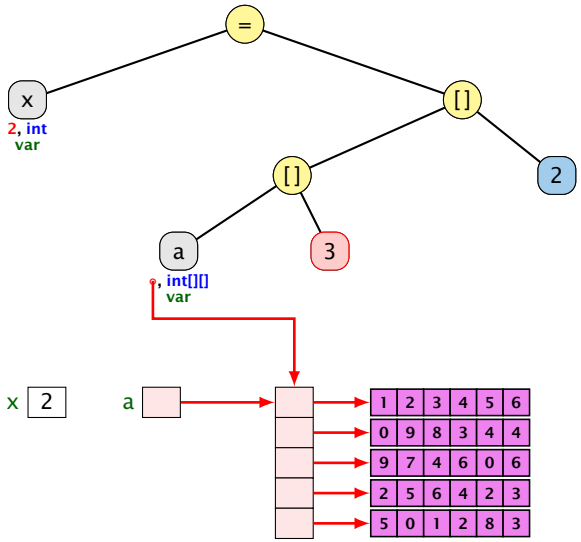


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

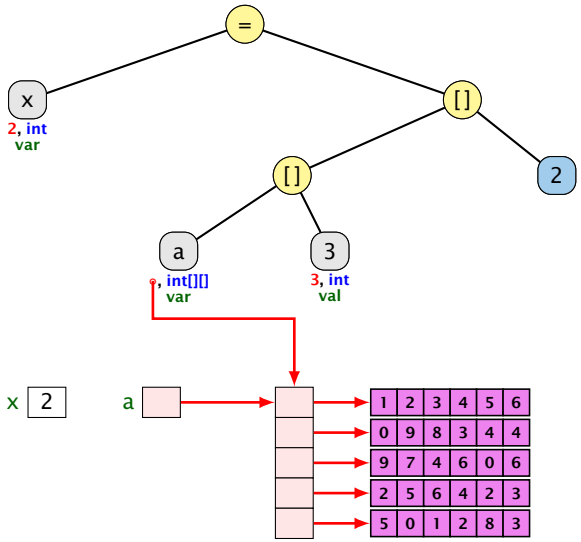


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

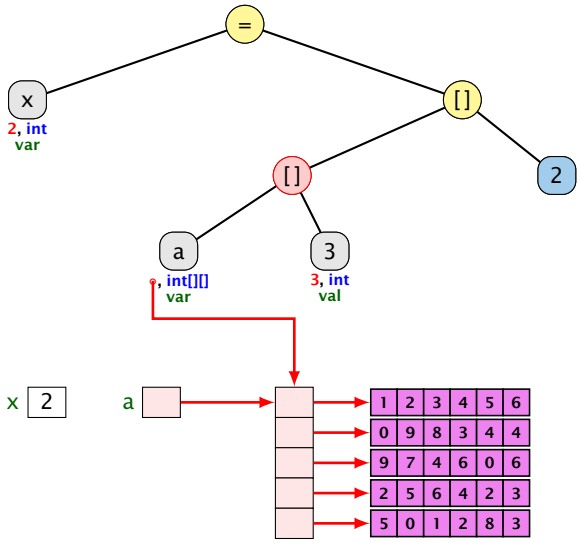


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

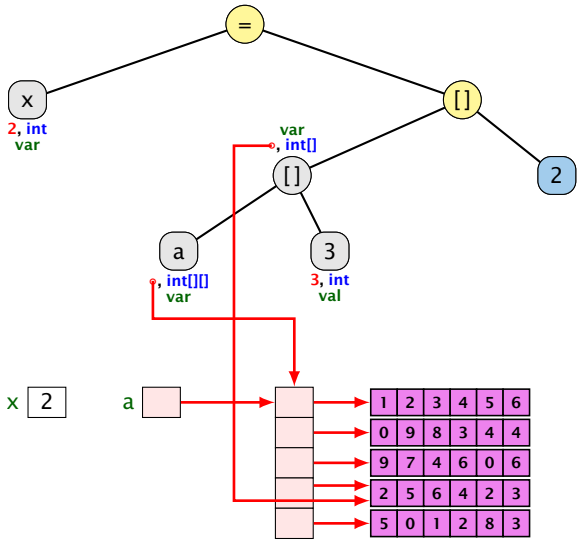


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

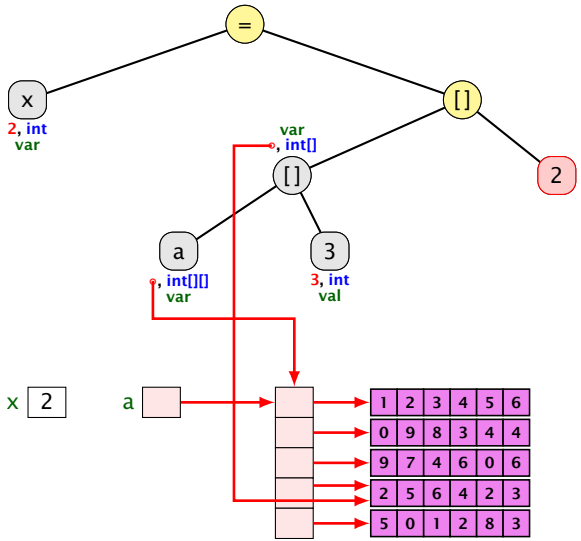


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

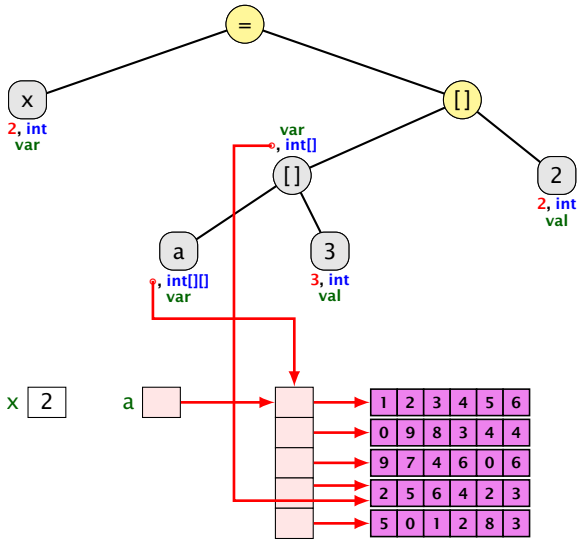


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

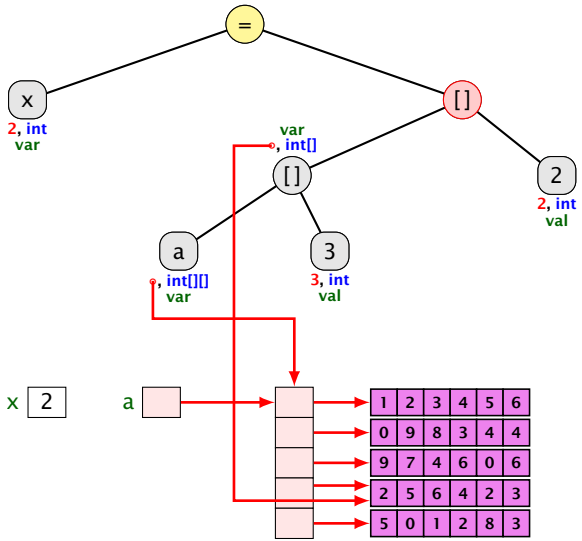


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

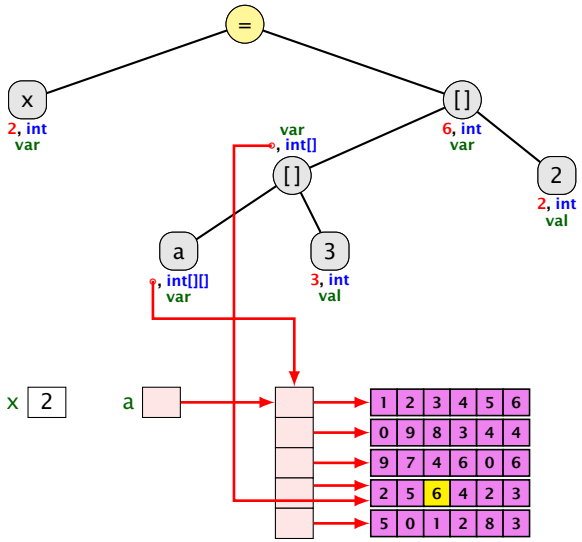


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

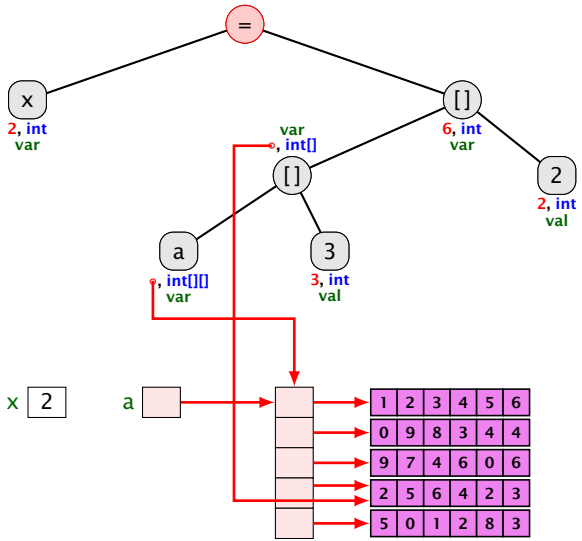


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]

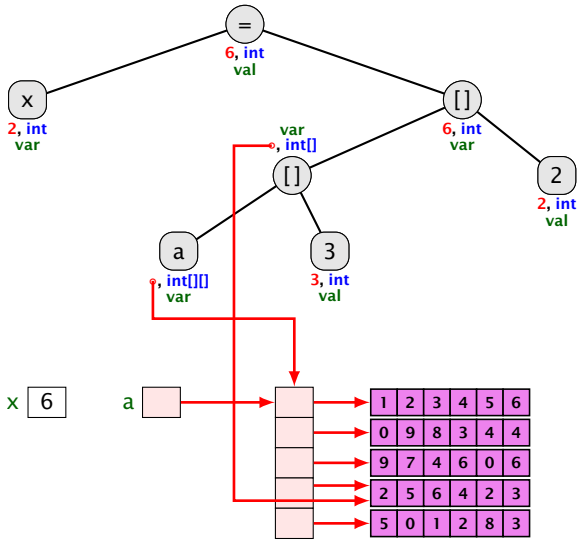


Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Beispiel: x = a[3][2]



Der Index-Operator

symbol	*name*	*types*	*L/R*	*level*
[]	index	array, int	links	1

Zugriff auf ein Arrayelement.

Der .-Operator

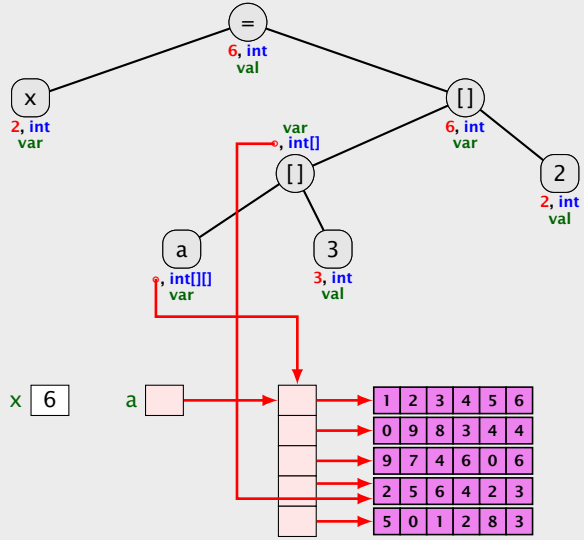
symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: x = a[3][2]



Beispiel: new int[2][4].length

Das Parsing für den new-Operator passt nicht in das Schema:

Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: `new int[2][4].length`

Das Parsing für den new-Operator passt nicht in das Schema:

Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: new int[2][4].length

Das Parsing für den new-Operator passt nicht in das Schema:

Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: `new int[2][4].length`

Das Parsing für den new-Operator passt nicht in das Schema:

Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: new int[2][4].length

Das Parsing für den new-Operator passt nicht in das Schema:

Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

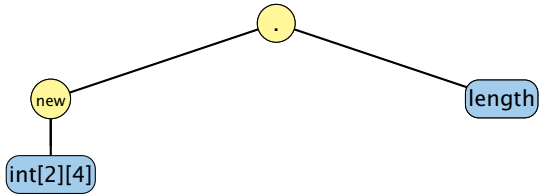
Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Beispiel: new int[2][4].length

Das Parsing für den new-Operator passt nicht in das Schema:



Der .-Operator

symbol	*name*	*types*	*L/R*	*level*
.	member access	Array/Objekt/Class, Member	links	1

Zugriff auf Member.

Beispiel:

- `x = new int[2][4].length`
  `x` hat dann den Wert 2.

Arrayinitialisierung

1. `int[] a = new int[3];`
   `a[0] = 1; a[1] = 2; a[2] = 3;`

2. `int[] a = new int[]{ 1, 2, 3};`

3. `int[] a = new int[3]{ 1, 2, 3};`

4. `int[] a = { 1, 2, 3};`

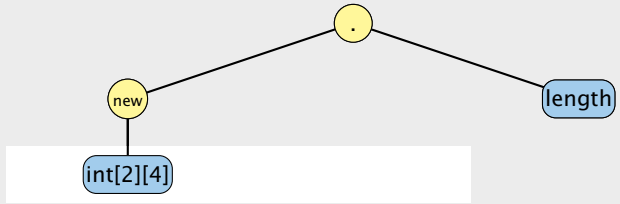
5. `char[][] b = { {'a','b'}, new char[3], {} };`

6. `char[][] b;`
   `b = new char[][]{ {'a','b'}, new char[3], {} };`

7. `char[][] b;`
   `b = { {'a','b'}, new char[3], {} };`

Beispiel: new int[2][4].length

Das Parsing für den new-Operator passt nicht in das Schema:



5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel

```
1 int result = a[0];
2 int i = 1;        // Initialisierung
3 while (i < a.length) {
4     if (a[i] < result)
5         result = a[i];
6     i = i+1;     // Modifizierung
7 }
8 write(result);
```

Bestimmung des Minimums

Beispiel

```
1 int result = a[0];
2 int i = 1;        // Initialisierung
3 while (i < a.length) {
4     if (a[i] < result)
5         result = a[i];
6     i = i+1;     // Modifizierung
7 }
8 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
1 int result = a[0];
2 for (int i = 1; i < a.length; ++i)
3     if (a[i] < result)
4         result = a[i];
5 write(result);
```

Bestimmung des Minimums

Das For-Statement

```
for (init, cond, modify) stmt
```

entspricht:

```
{ init; while (cond) { stmt modify; } }
```

Erläuterungen:

- `++i;` ist äquivalent zu `i = i + 1;`
- die `while`-Schleife steht innerhalb eines Blocks (`{...}`)

  die Variable `i` ist außerhalb dieses Blocks nicht sichtbar/zugreifbar

Das For-Statement

```
1 int result = a[0];
2 for (int i = 1; i < a.length; ++i)
3     if (a[i] < result)
4         result = a[i];
5 write(result);
```

Bestimmung des Minimums

5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel

```
public static int[] readArray(int number) {
    // number = Anzahl zu lesender Elemente
    int[] result = new int[number]; // Feld anlegen
    for (int i = 0; i < number; ++i) {
        result[i] = read();
    }
    return result;
}
```

Einlesen eines Feldes

5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel

Type des Rückgabewertes

```
public static int[] readArray(int number) {
    // number = Anzahl zu lesender Elemente
    int[] result = new int[number]; // Feld anlegen
    for (int i = 0; i < number; ++i) {
        result[i] = read();
    }
    return result;
}
```

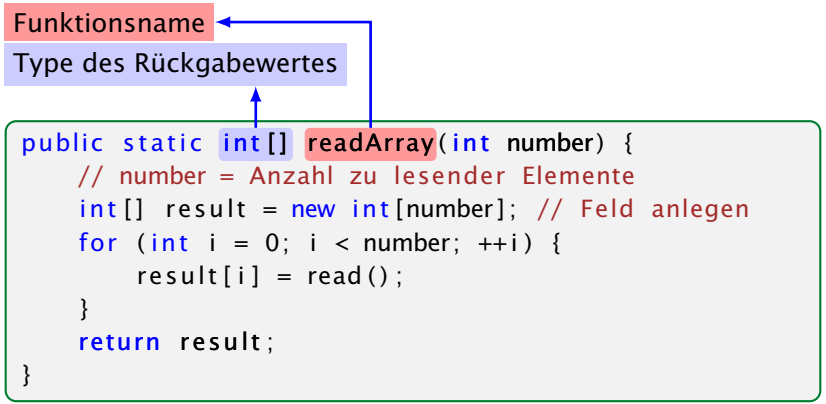
Einlesen eines Feldes

5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel



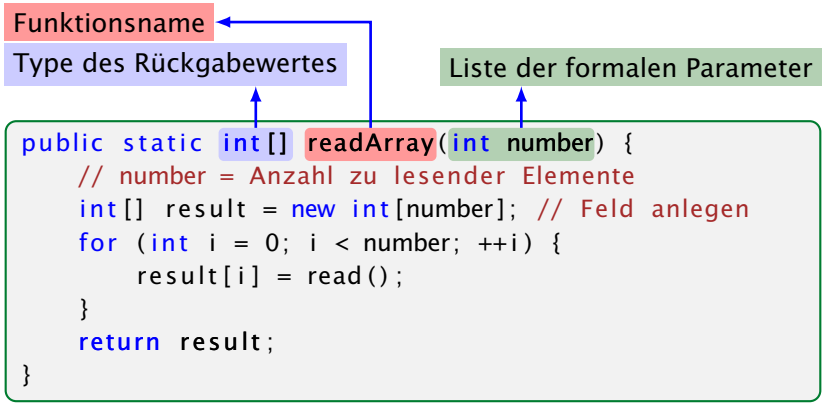
Einlesen eines Feldes

5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel

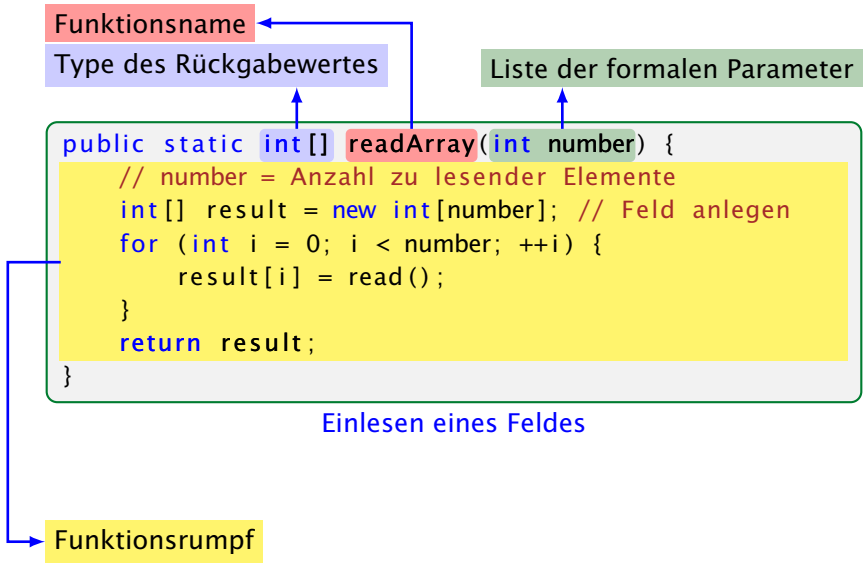


Einlesen eines Feldes

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel

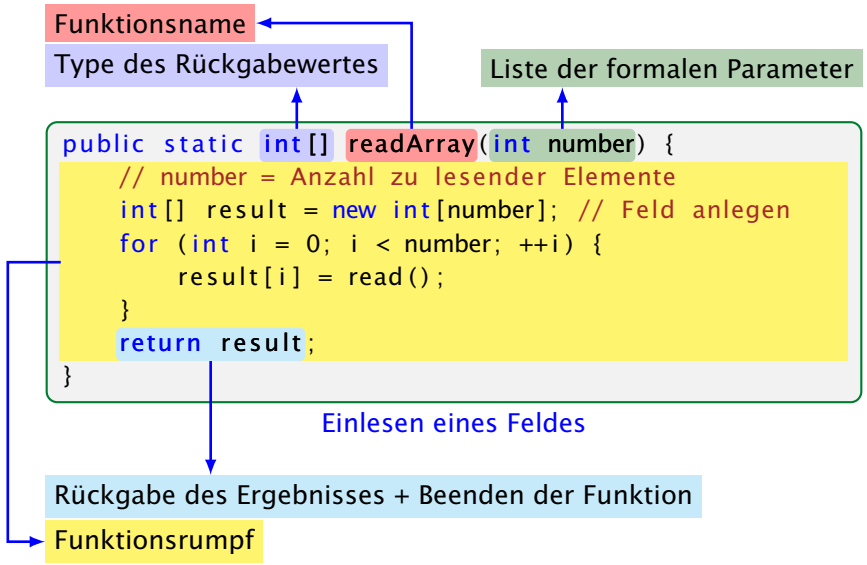


5.6 Funktionen und Prozeduren

Oft möchte man:

- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

Beispiel



5.6 Funktionen und Prozeduren

Oft möchte man:

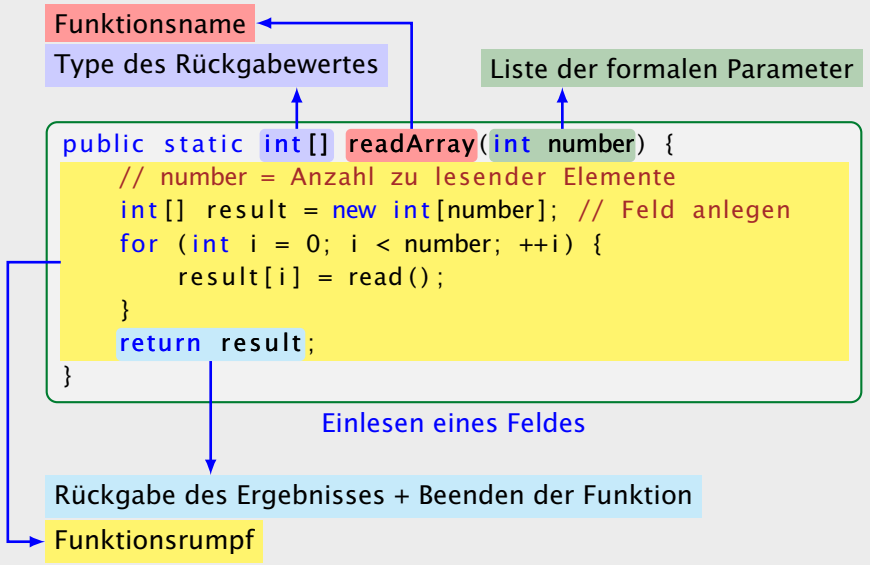
- Teilprobleme separat lösen; und dann
- die Lösung mehrfach verwenden.

5.6 Funktionen und Prozeduren

Erläuterungen:

- Die erste Zeile ist der Header der Funktion.
- `public`, und `static` komen später
- `int[]` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der formalen Parameter, hier: `(int number)`.
- Der Rumpf der Funktion steht in geschweiften Klammern.
- `return expr;` beendet die Ausführung der Funktion und liefert den Wert von `expr` zurück.

Beispiel



5.6 Funktionen und Prozeduren

Erläuterungen:

- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von `'{'` und `'}'`, sind nur innerhalb dieses Blocks sichtbar, d.h. benutzbar (lokale Variablen).
- Der Rumpf einer Funktion ist ein Block.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert 7 (aktueller Parameter).

5.6 Funktionen und Prozeduren

Erläuterungen:

- Die erste Zeile ist der Header der Funktion.
- `public`, und `static` komen später
- `int[]` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der formalen Parameter, hier: `(int number)`.
- Der Rumpf der Funktion steht in geschweiften Klammern.
- `return expr;` beendet die Ausführung der Funktion und liefert den Wert von `expr` zurück.

Beispiel

```
public static int min(int[] b) {
    int result = b[0];
    for (int i = 1; i < b.length; ++i) {
        if (b[i] < result)
            result = b[i];
    }
    return result;
}
```

Bestimmung des Minimums

5.6 Funktionen und Prozeduren

Erläuterungen:

- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von '{' und '}', sind nur innerhalb dieses Blocks sichtbar, d.h. benutzbar (lokale Variablen).
- Der Rumpf einer Funktion ist ein Block.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `number` den Wert 7 (aktueller Parameter).

Beispiel

```
public class Min extends MiniJava {
    public static int[] readArray(int number) { ... }
    public static int min(int[] b) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main(String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

Programm zur Minimumsberechnung

Beispiel

```
public static int min(int[] b) {
    int result = b[0];
    for (int i = 1; i < b.length; ++i) {
        if (b[i] < result)
            result = b[i];
    }
    return result;
}
```

Bestimmung des Minimums

Beispiel

```
public class Min extends MiniJava {
    public static int[] readArray(int number) { ... }
    public static int min(int[] b) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main(String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    } // end of main()
} // end of class Min
```

Programm zur Minimumsberechnung

Beispiel

Erläuterungen:

- Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen Prozeduren.
- Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {
    public static void main (String[] args) {
        write(args[0]+args[1]);
    }
} // end of class Test
```

Beispiel

Erläuterungen:

- Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen Prozeduren.
- Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {
    public static void main (String[] args) {
        write(args[0]+args[1]);
    }
} // end of class Test
```


Beispiel

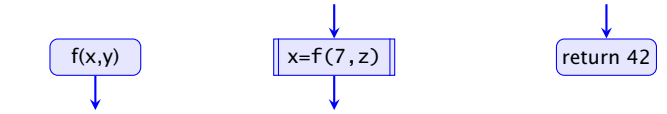
Der Aufruf

```
java Test "Hel" "lo World!"
```

liefert: `Hello World!`

5.6 Funktionen und Prozeduren

Um die Arbeitsweise von Funktionen zu veranschaulichen erweitern/modifizieren wir die Kontrollflussdiagramme



- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.

Beispiel

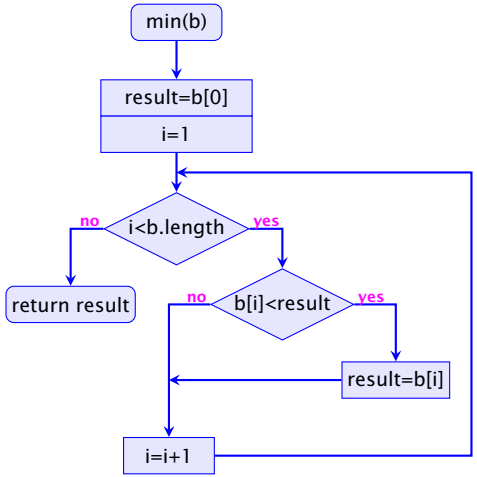
Der Aufruf

```
java Test "Hel" "lo World!"
```

liefert: `Hello World!`

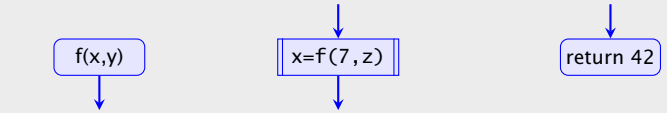
5.6 Funktionen und Prozeduren

Teildiagramm der Funktion `min()`:



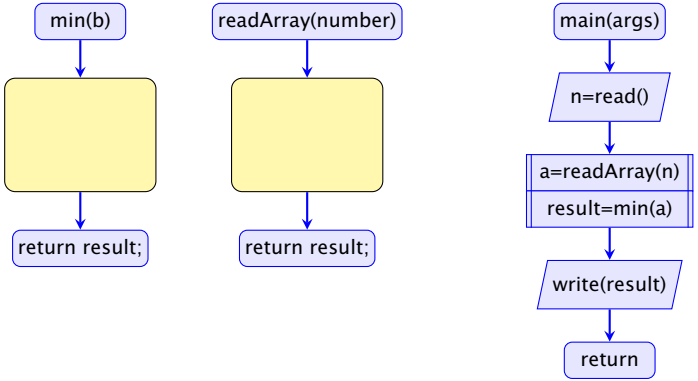
5.6 Funktionen und Prozeduren

Um die Arbeitsweise von Funktionen zu veranschaulichen erweitern/modifizieren wir die Kontrollflussdiagramme



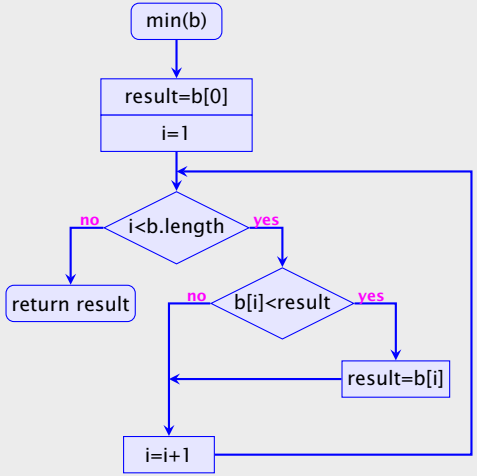
Startknoten der Funktion f | Funktionsaufruf | Endknoten

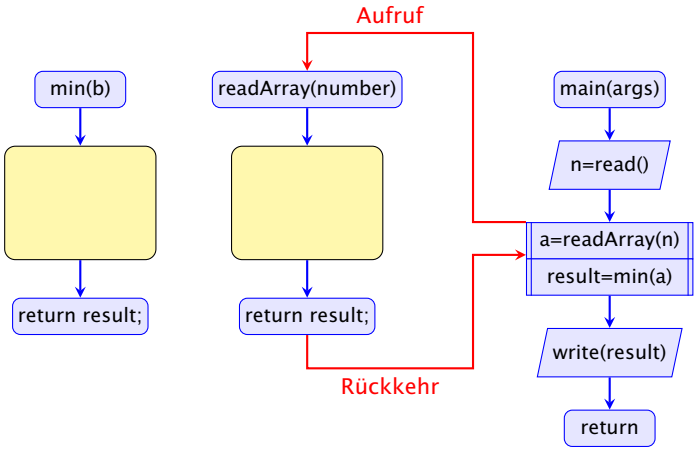
- Für jede Funktion wird ein eigenes Teildiagramm erstellt.
- Ein Aufrufknoten repäsentiert eine Teilberechnung der aufgerufenen Funktion.



5.6 Funktionen und Prozeduren

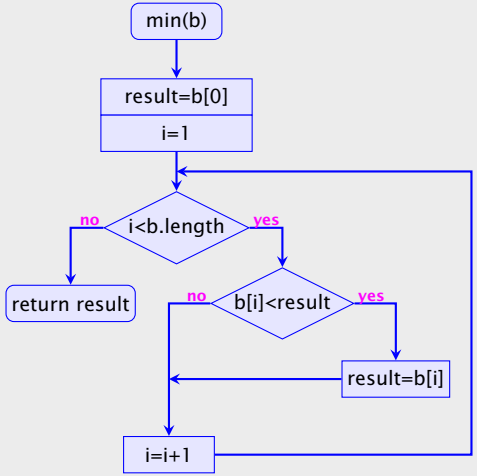
Teildiagramm der Funktion `min()`:



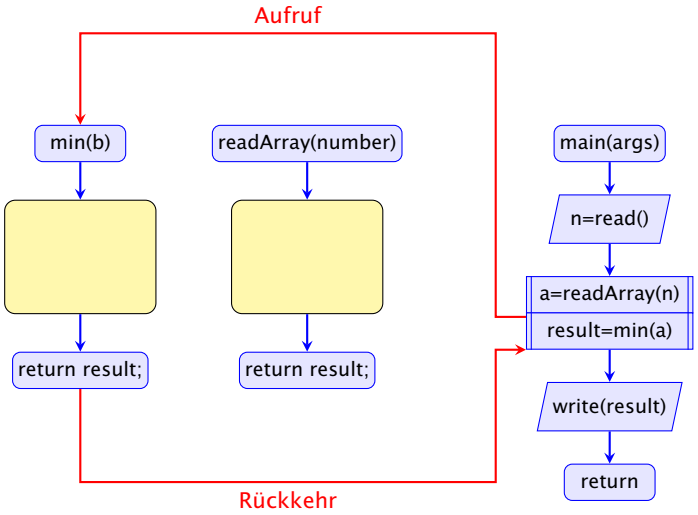


5.6 Funktionen und Prozeduren

Teildiagramm der Funktion `min()`:

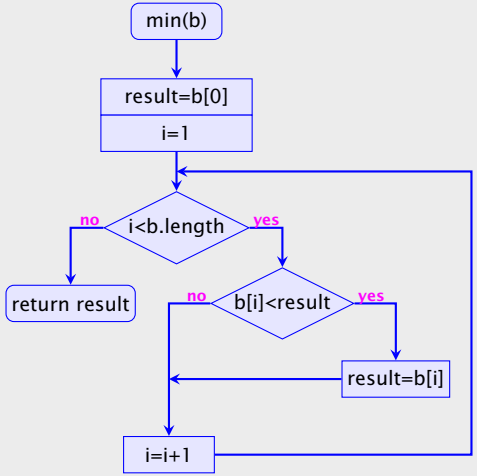


5.6 Funktionen und Prozeduren



5.6 Funktionen und Prozeduren

Teildiagramm der Funktion `min()`:

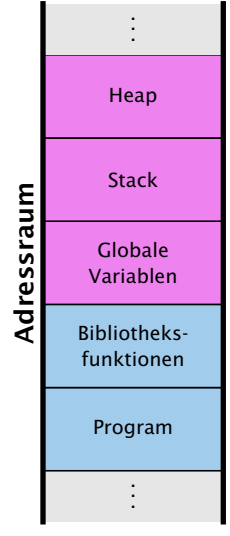


6 Speicherorganisation

Der Speicher des Programms ist in verschiedene Speicherbereiche untergliedert

- Speicherbereiche, die den eigentlichen Programmcode und den Code der Laufzeitbibliothek enthalten;
- einen Speicherbereich für globale/statische Variablen;
- einen Speicherbereich Heap, und
- einen Speicherbereich Stack.

Variablen werden üblicherweise auf dem Heap oder dem Stack gespeichert.



Heap vs. Stack vs. statisch

Heap

Auf dem Heap können zur Laufzeit zusammenhängende Speicherbereiche angefordert werden, und in beliebiger Reihenfolge wieder freigegeben werden.

Stack

Der Stack ist ein Speicherbereich, auf dem neue Elemente oben gespeichert werden, und Freigaben in umgekehrter Reihenfolge (d.h. oben zuerst) erfolgen müssen (LIFO = Last In First Out).

Statische Variablen

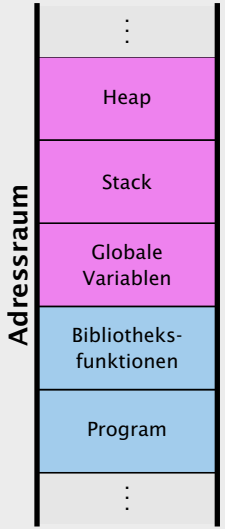
Statische Variablen werden zu Beginn des Programms angelegt, und zum Ende des Programms wieder gelöscht.

6 Speicherorganisation

Der Speicher des Programms ist in verschiedene Speicherbereiche untergliedert

- Speicherbereiche, die den eigentlichen Programmcode und den Code der Laufzeitbibliothek enthalten;
- einen Speicherbereich für globale/statische Variablen;
- einen Speicherbereich Heap, und
- einen Speicherbereich Stack.

Variablen werden üblicherweise auf dem Heap oder dem Stack gespeichert.



Statische Variablen

Statische Variablen (auch Klassenvariablen) werden im Klassenrumpf ausserhalb einer Funktion definiert.

Jede Funktion der Klasse kann dann diese Variablen benutzen; deshalb werden sie manchmal auch globale Variablen genannt.

Heap vs. Stack vs. statisch

Heap
Auf dem Heap können zur Laufzeit zusammenhängende Speicherbereiche angefordert werden, und in beliebiger Reihenfolge wieder freigegeben werden.

Stack
Der Stack ist ein Speicherbereich, auf dem neue Elemente oben gespeichert werden, und Freigaben in umgekehrter Reihenfolge (d.h. oben zuerst) erfolgen müssen (LIFO = Last In First Out).

Statische Variablen
Statische Variablen werden zu Beginn des Programms angelegt, und zum Ende des Programms wieder gelöscht.

Statische Variablen

Statische Variablen (auch Klassenvariablen) werden im Klassenrumpf ausserhalb einer Funktion definiert.

Jede Funktion der Klasse kann dann diese Variablen benutzen; deshalb werden sie manchmal auch globale Variablen genannt.

Beispiel – Statische Variablen

```
1  public class GGT extends MiniJava {
2      static int x, y;
3      static void readInput() {
4          x = read();
5          y = read();
6      }
7      public static void main (String[] args) {
8          readInput();
9          while (x != y) {
10             if (x < y)
11                 y = y - x;
12             else
13                 x = x - y;
14         }
15         write(x);
16     }
17 }
```

Beispiel – Statische Variablen

```
1  public class GGT extends MiniJava {
2      static int x, y;
3      static void readInput() {
4          x = read();
5          y = read();
6      }
7      public static void main (String[] args) {
8          readInput();
9          while (x != y) {
10             if (x < y)
11                 y = y - x;
12             else
13                 x = x - y;
14         }
15         write(x);
16     }
17 }
```


Verwendung des Heaps

Speicherallokation mit dem Operator new:

```
int[][] arr;
arr = new int[10][]; // array mit int-Verweisen
```

Immer wenn etwas mit new angelegt wird, landet es auf dem Heap.

Wenn keine Referenz mehr auf den angeforderten Speicher existiert kann der Garbage Collector den Speicher freigeben:

```
int[][] arr;
arr = new int[10][]; // array mit int-Verweisen
arr = null; // jetzt koennte GC freigeben
```

Verwendung des Heaps

Speicherallokation mit dem Operator new:

```
int[][] arr;
arr = new int[10][]; // array mit int-Verweisen
```

Immer wenn etwas mit new angelegt wird, landet es auf dem Heap.

Wenn keine Referenz mehr auf den angeforderten Speicher existiert kann der Garbage Collector den Speicher freigeben:

```
int[][] arr;
arr = new int[10][]; // array mit int-Verweisen
arr = null; // jetzt koennte GC freigeben
```


Verwendung des Heaps

Beispiel:

```
1  public static int[] readArray(int number) {
2      // number = Anzahl zu lesender Elemente
3      int[] result = new int[number];
4      for (int i = 0; i < number; ++i) {
5          result[i] = read();
6      }
7      return result;
8  }
9  public static void main(String[] args) {
10     readArray(6);
11 }
```

Da die von readArray zurückgegebene Referenz nicht benutzt wird, kann der GC freigeben.

Verwendung des Heaps

Beispiel:

```
1 public static void main(String[] args) {
2     int[] b = readArray(6);
3     int[] c = b;
4     b = null;
5 }
```

Da c immer noch eine Referenz auf das array enthält erfolgt keine Freigabe.

Verwendung des Heaps

Beispiel:

```
1  public static int[] readArray(int number) {
2      // number = Anzahl zu lesender Elemente
3      int[] result = new int[number];
4      for (int i = 0; i < number; ++i) {
5          result[i] = read();
6      }
7      return result;
8  }
9  public static void main(String[] args) {
10     readArray(6);
11 }
```

Da die von readArray zurückgegebene Referenz nicht benutzt wird, kann der GC freigeben.

Verwendung des Stacks

- Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

Verwendung des Heaps

Beispiel:

```
1 public static void main(String[] args) {
2     int[] b = readArray(6);
3     int[] c = b;
4     b = null;
5 }
```

Da c immer noch eine Referenz auf das array enthält erfolgt keine Freigabe.

Verwendung des Stacks

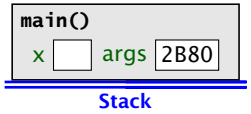
- Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Verwendung des Stacks

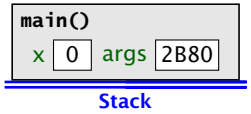
- Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Verwendung des Stacks

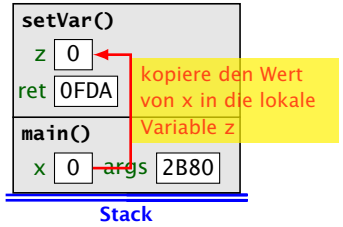
- Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Verwendung des Stacks

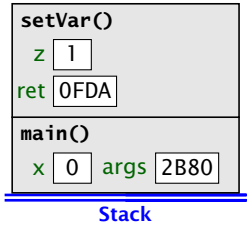
- Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
    z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Verwendung des Stacks

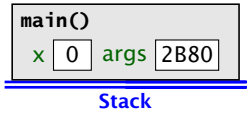
- Bei Aufruf einer Funktion (auch `main()`) werden lokale Variablen (d.h. auch Werte von aktuellen Parametern) und die Rücksprungadresse als Frames auf dem Stack gespeichert.
- Während der Programmausführung sind nur die Variablen im obersten Frame zugreifbar.
- Bei der Beendigung einer Funktion wird der zugehörige Stackframe gelöscht.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



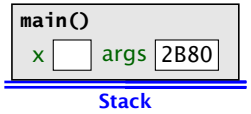
Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



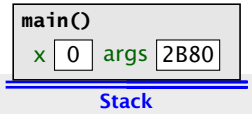
Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



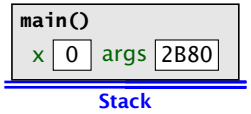
Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



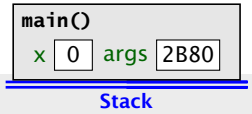
Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



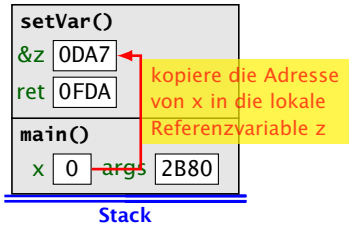
Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



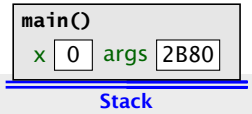
Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



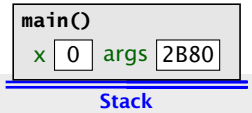
Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Parameterübergabe – Call-by-Value

```
public static void setVar(int z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



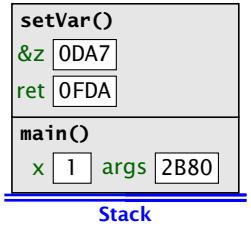
Das ist die einzige Form der Parameterübergabe, die Java unterstützt.

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



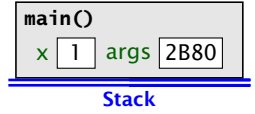
Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```

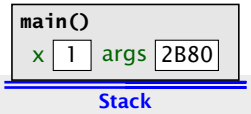
Ausgabe: `arr[1] == 1`

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

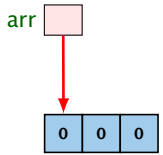
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



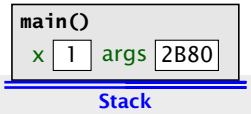
Ausgabe: arr[1] == 1

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

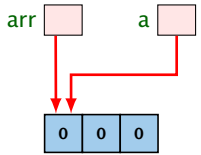
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



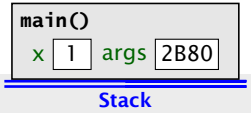
Ausgabe: arr[1] == 1

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

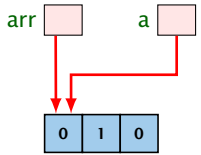
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



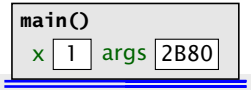
Ausgabe: arr[1] == 1

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

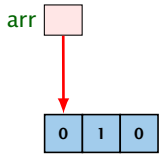
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



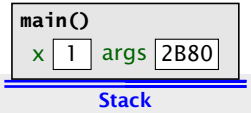
Ausgabe: arr[1] == 1

Parameterübergabe – Call-by-Reference

```
public static void setVar(int &z) {
     z = 1;
}

public static void main(String[] args) {
    int x;
    x = 0;
    setVar(x);

    write("x == " + x);
}
```



Diese Form der Parameterübergabe ist in Java nicht möglich, aber z.B. in C++.

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```

Ausgabe: `arr[1] == 0`

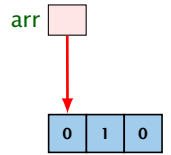
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



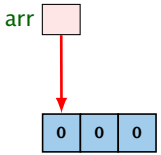
Ausgabe: `arr[1] == 1`

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: arr[1] == 0

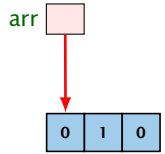
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



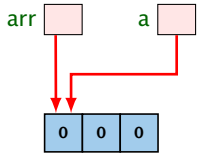
Ausgabe: arr[1] == 1

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: arr[1] == 0

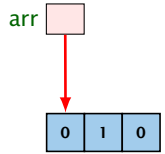
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



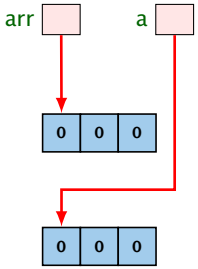
Ausgabe: arr[1] == 1

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

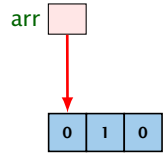
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



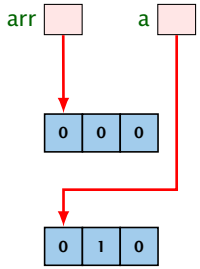
Ausgabe: `arr[1] == 1`

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: arr[1] == 0

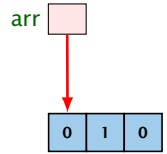
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



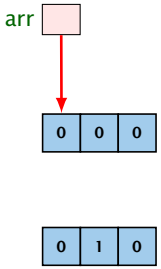
Ausgabe: arr[1] == 1

Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: arr[1] == 0

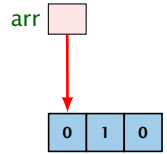
Parameterübergabe – Referenzvariablen

Auch Referenzvariablen werden per call-by-value übergeben. Man kann den Inhalt des zugehörigen Objekts/Arrays aber verändern.

```
public static void setVar(int[] a) {
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];

    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: arr[1] == 1

Rekursive Funktionen

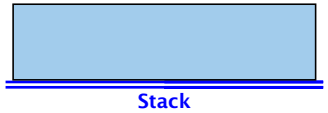
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

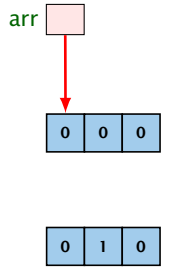


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

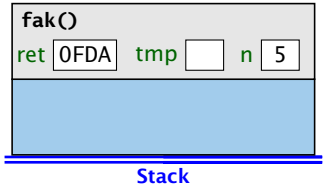
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

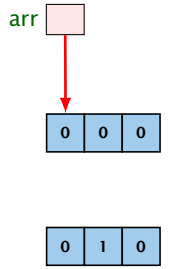


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

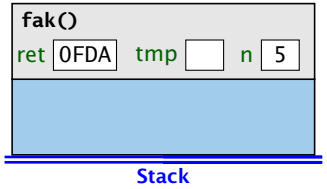
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

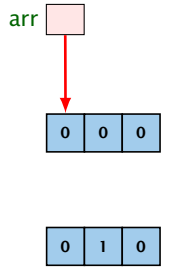


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

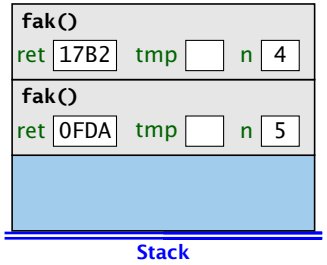
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

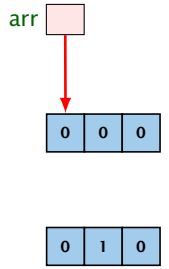


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
    a    = new int[3];
    a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

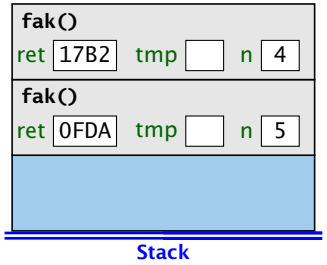
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

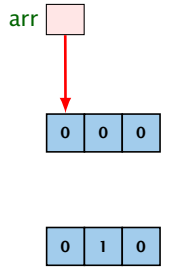


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

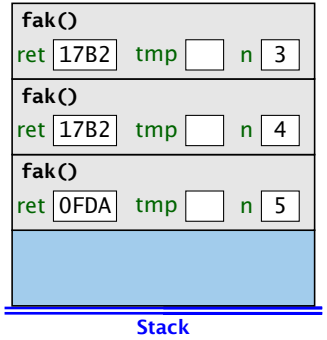
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

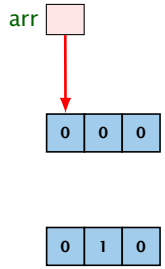


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

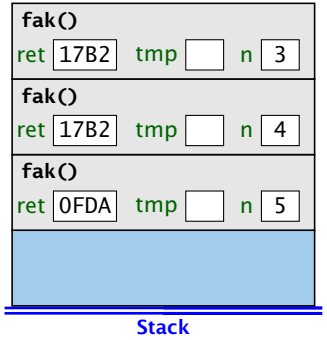
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

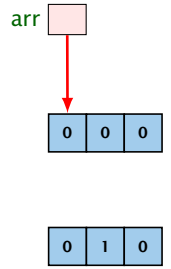


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
    a    = new int[3];
    a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

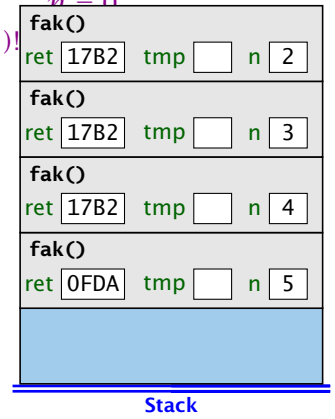
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

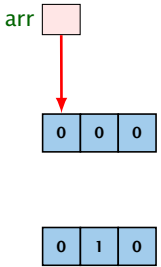


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

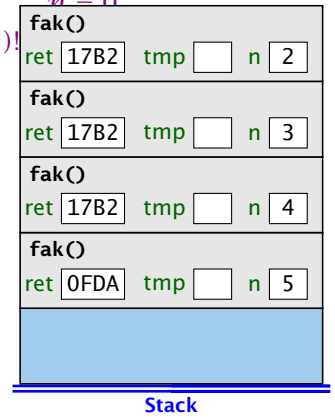
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

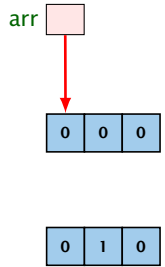


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

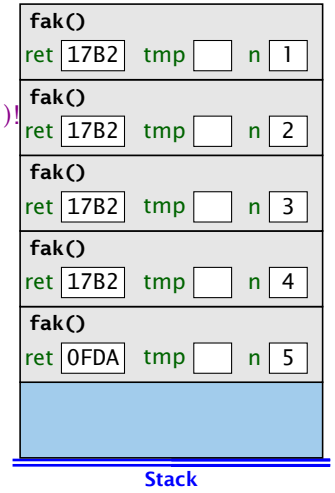
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

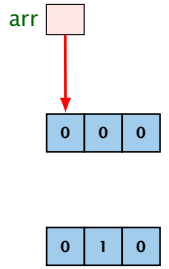


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

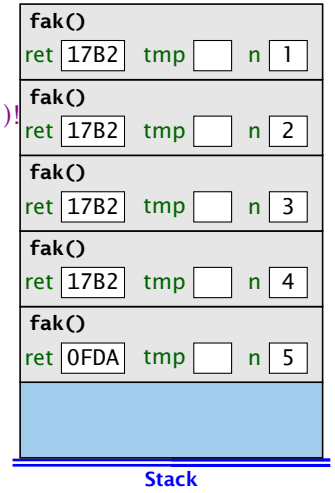
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

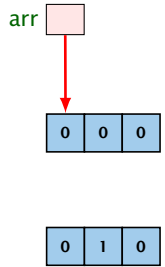


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

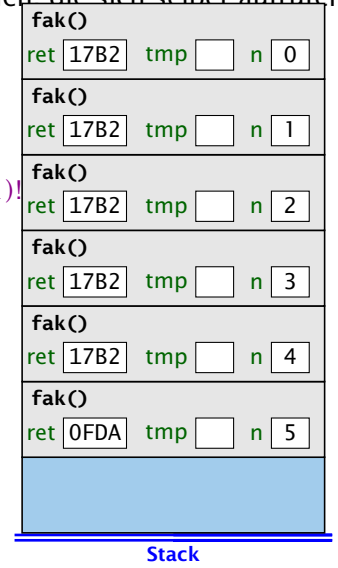
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

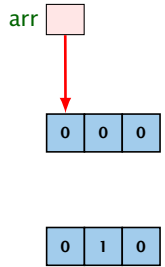


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

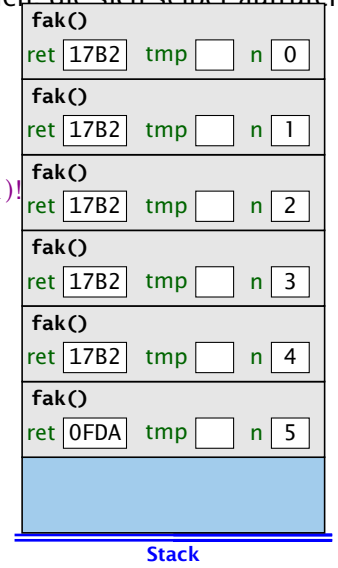
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

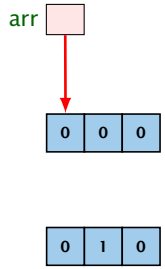


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

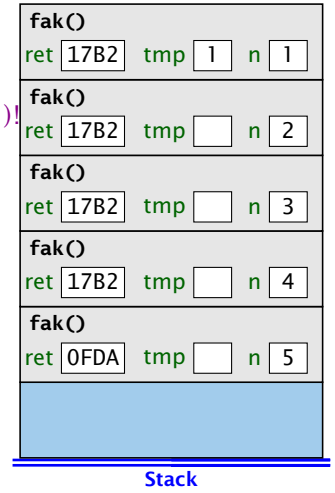
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

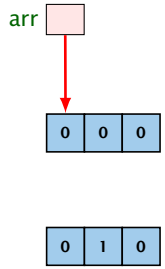


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

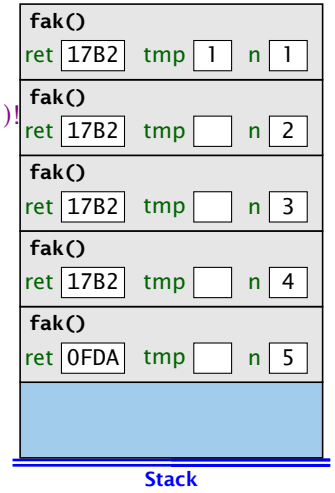
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 \\ n \cdot (n-1)! \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

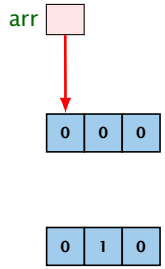


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

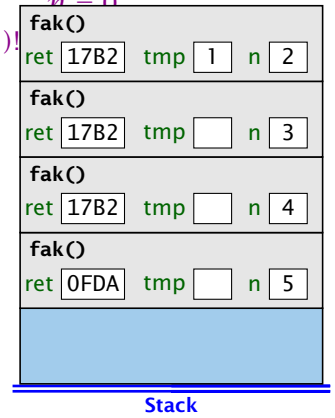
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

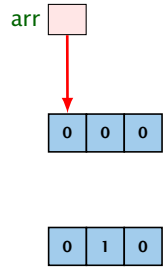


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

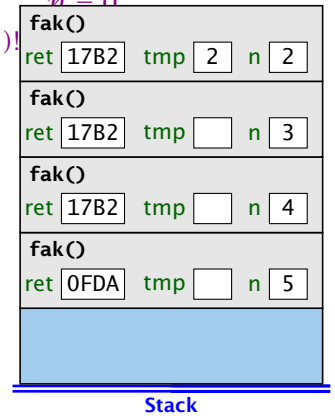
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

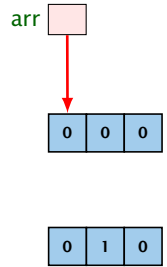


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

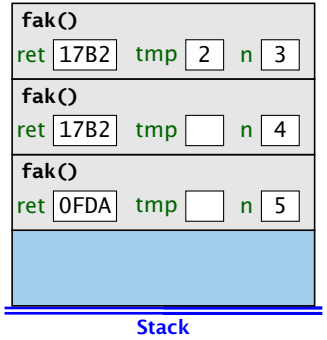
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

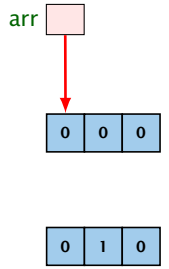


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

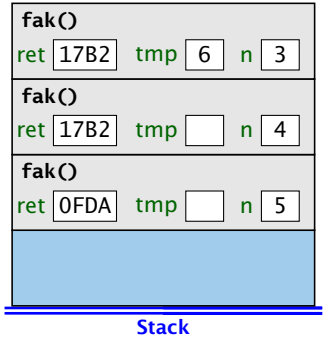
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

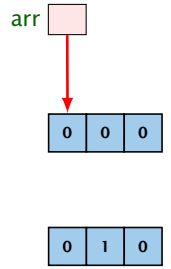


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

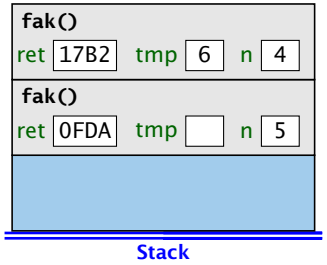
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

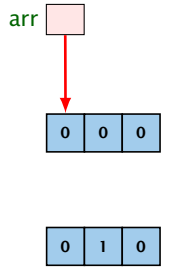


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

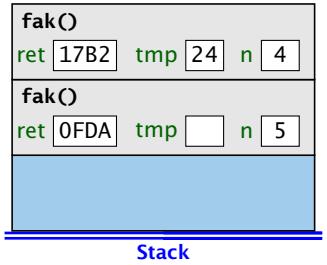
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

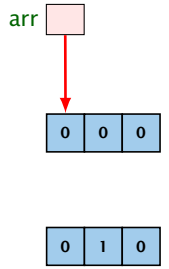


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

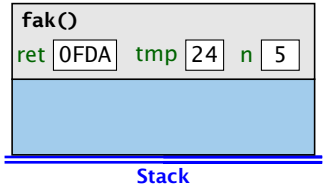
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

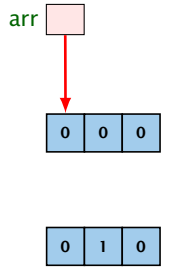


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Rekursive Funktionen

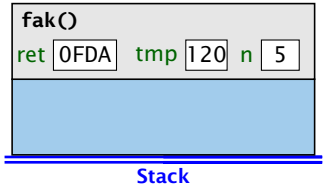
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```

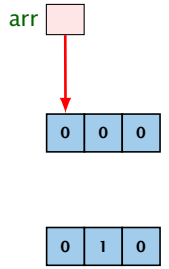


Parameterübergabe – Referenzvariablen

Wenn man das Objekt selber verändert, ist die Änderung nicht nach aussen sichtbar.

```
public static void setVar(int[] a) {
     a    = new int[3];
     a[1] = 1;
}

public static void main(String[] args) {
    // initialize array elements to 0
    int[] arr = new int[3];
    setVar(arr);
    write("arr[1] == " + arr[1]);
}
```



Ausgabe: `arr[1] == 0`

Vollständiger Code

```
1  public class Fakultaet {
2      public static long fak(int n) {
3          if (n > 0)
4              return n*fak(n-1);
5          else
6              return 1;
7      }
8      public static void main(String args[]) {
9          System.out.println(fak(20));
10     }
11 }
```


Rekursive Funktionen

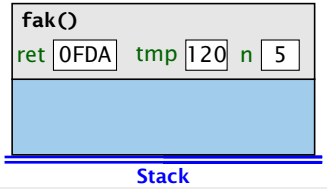
Rekursive Funktionen sind Funktionen, die sich selber aufrufen (eventuell über Umwege).

Beispiel: Fakultätsberechnung

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

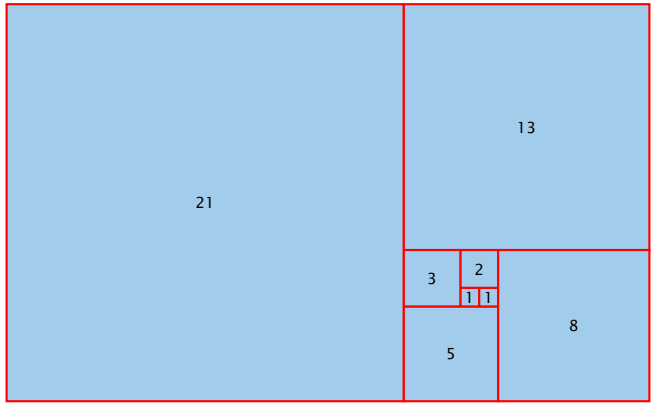
```
public static long fak(int n) {
    long tmp;

    if (n > 0) {
        tmp  = fak(n-1);
        tmp *= n;
        return tmp;
    }
    else return 1;
}
```



Fibonaccizahlen

$$F_n = \begin{cases} n & 0 \le n \le 1 \\ F_{n-1} + F_{n-2} & n \ge 2 \end{cases}$$



Vollständiger Code

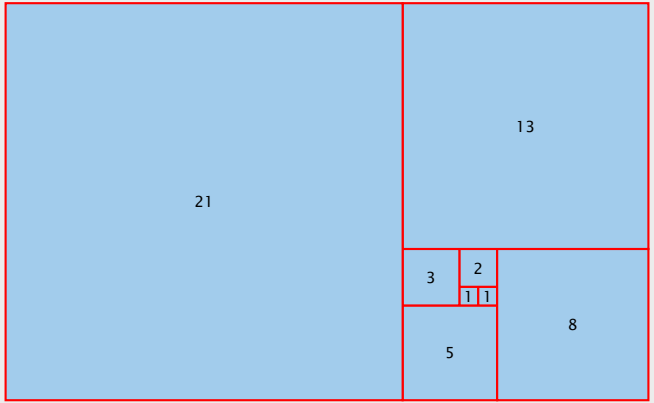
```
1  public class Fakultaet {
2      public static long fak(int n) {
3          if (n > 0)
4              return n*fak(n-1);
5          else
6              return 1;
7      }
8      public static void main(String args[]) {
9          System.out.println(fak(20));
10     }
11 }
```

Vollständiger Code

```
1  public class Fibonacci {
2      public static long fib(int n) {
3          if (n > 1)
4              return fib(n-1)+fib(n-2);
5          else
6              return n;
7      }
8
9      public static void main(String args[]) {
10         System.out.println(fib(50));
11     }
12 }
```


Fibonaccizahlen

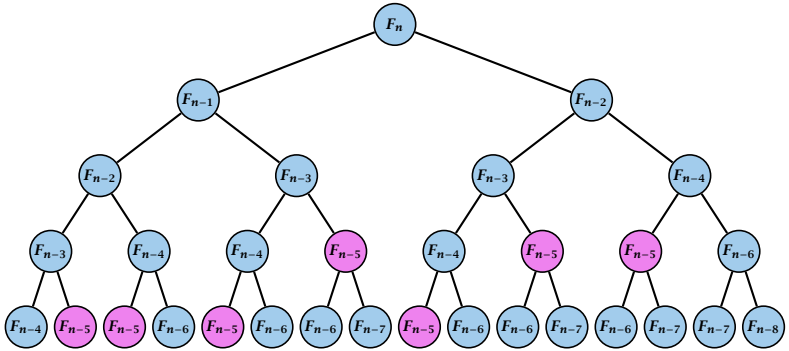
$$F_n = \begin{cases} n & 0 \le n \le 1 \\ F_{n-1} + F_{n-2} & n \ge 2 \end{cases}$$



Fibonaccizahlen

Programmlauf benötigt mehr als 1min.

Warum ist das so langsam?



Wir erzeugen viele rekursive Aufrufe für die gleichen Teilprobleme!

Vollständiger Code

```
1  public class Fibonacci {
2      public static long fib(int n) {
3          if (n > 1)
4              return fib(n-1)+fib(n-2);
5          else
6              return n;
7      }
8
9      public static void main(String args[]) {
10         System.out.println(fib(50));
11     }
12 }
```

Fibonaccizahlen

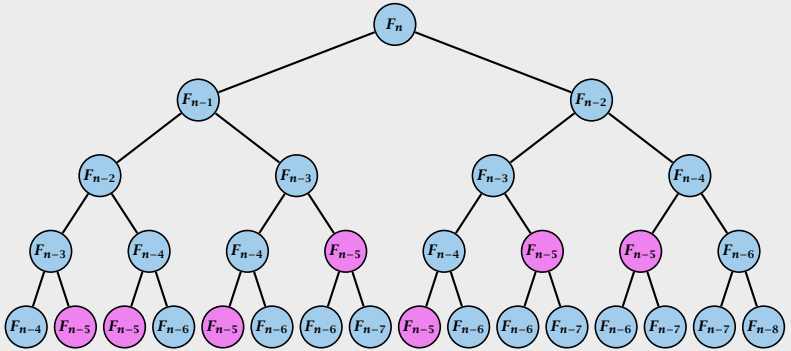
Lösung

- Speichere die Lösung für ein Teilproblem in einer globalen Variable.
- Wenn das Teilproblem das nächste mal gelöst werden soll braucht man nur nachzuschauen...

Fibonaccizahlen

Programmlauf benötigt mehr als 1min.

Warum ist das so langsam?



Wir erzeugen viele rekursive Aufrufe für die gleichen Teilprobleme!

Vollständiger Code

```
1  public class FibonacciImproved {
2      // F_93 does not fit into a long
3      static long[] lookup = new long[93];
4
5      public static long fib(int n) {
6          if (lookup[n] > 0) return lookup[n];
7
8          if (n > 1) {
9              lookup[n] = fib(n-1)+fib(n-2);
10             return lookup[n];
11         } else
12             return n;
13     }
14     public static void main(String args[]) {
15         System.out.println(fib(50));
16     }
17 }
```

Fibonaccizahlen

Lösung

- Speichere die Lösung für ein Teilproblem in einer globalen Variable.
- Wenn das Teilproblem das nächste mal gelöst werden soll braucht man nur nachzuschauen...

7 Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

Idee:

- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

$\Longrightarrow$ Sortieren durch Einfügen (↑InsertionSort)

7 Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

Idee:

- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!

$\Longrightarrow$ Sortieren durch Einfügen (↑InsertionSort)

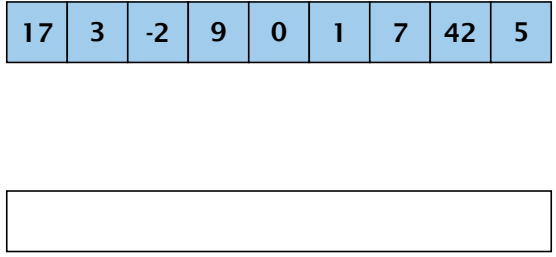
7 Anwendung: Sortieren

```
1  public static int[] sort(int[] a) {
2      int n = a.length;
3      int[] b = new int[n];
4      for (int i = 0; i < n; ++i)
5          insert(b, a[i], i);
6              // b    = Feld, in das eingefuegt wird
7              // a[i] = einzufuegendes Element
8              // i    = Anzahl von Elementen in b
9      return b;
10 } // end of sort ()
```

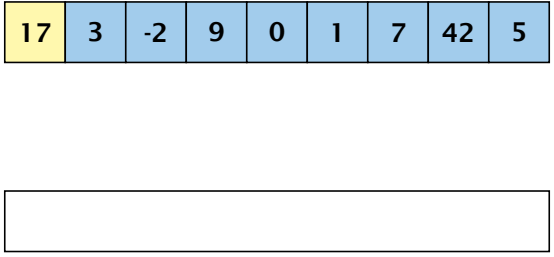
Sortieren durch Einfügen

Teilproblem: wie fügt man ein?

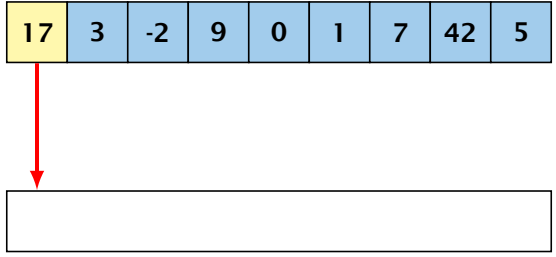
Beispiel



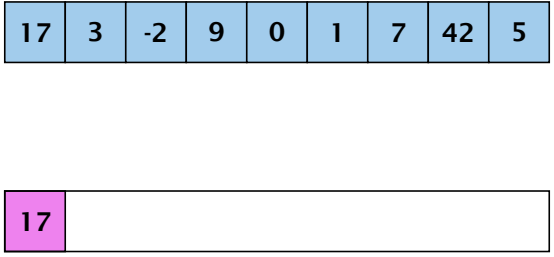
Beispiel



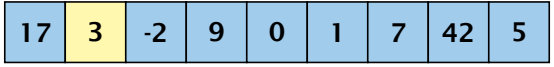
Beispiel

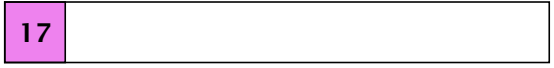


Beispiel

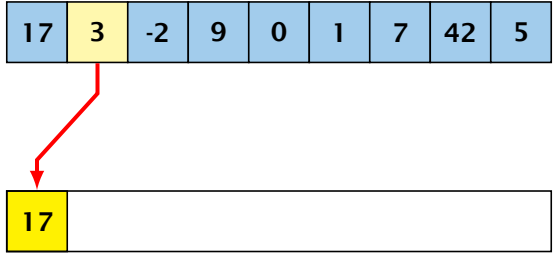


Beispiel

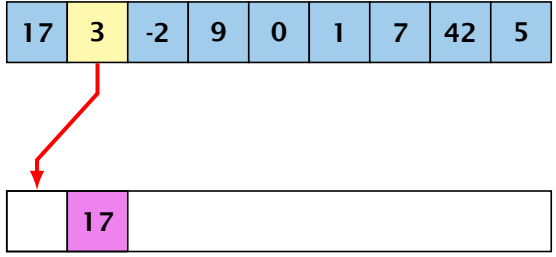




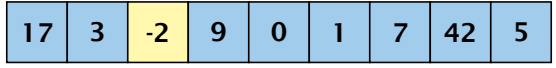
Beispiel

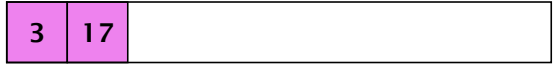


Beispiel

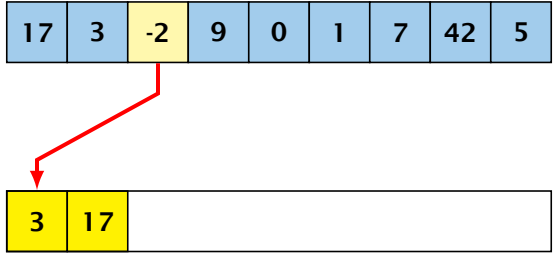


Beispiel

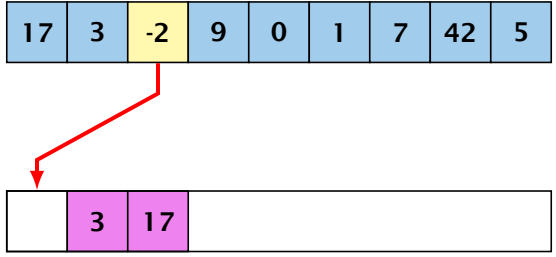




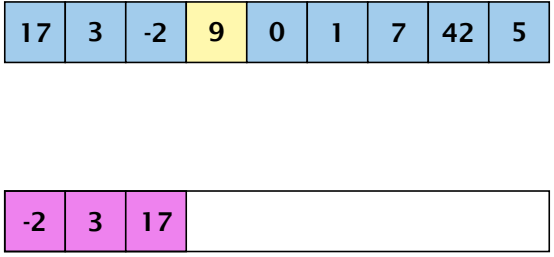
Beispiel



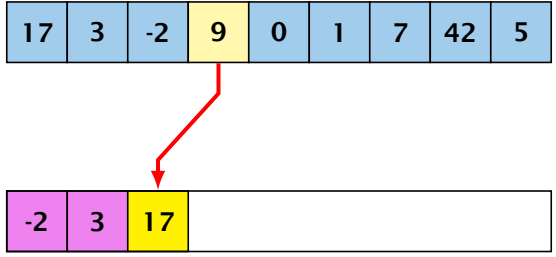
Beispiel



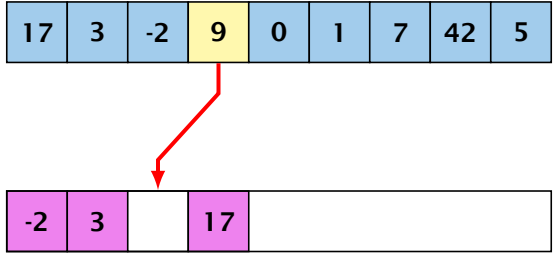
Beispiel



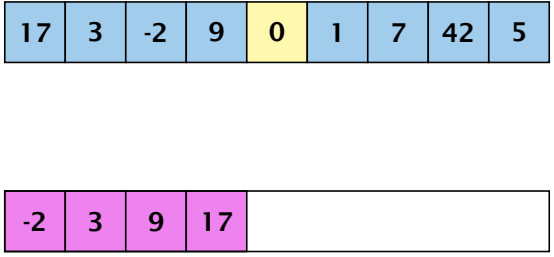
Beispiel



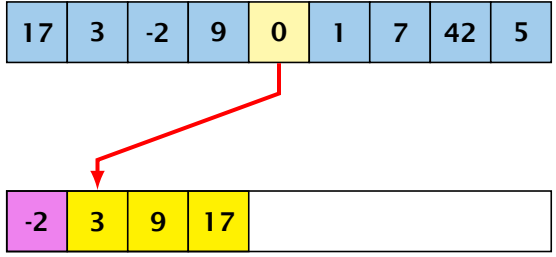
Beispiel



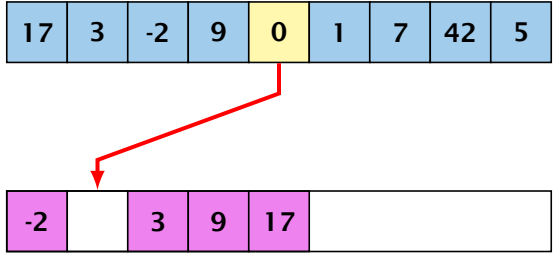
Beispiel



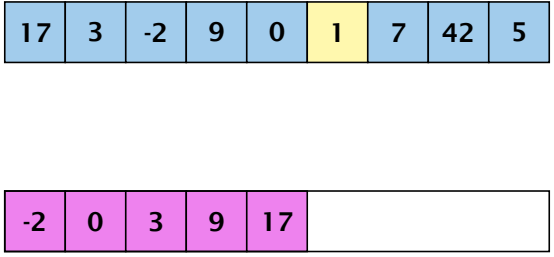
Beispiel



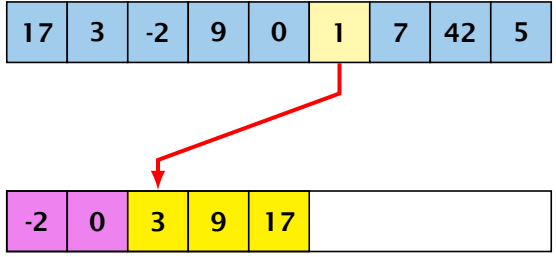
Beispiel



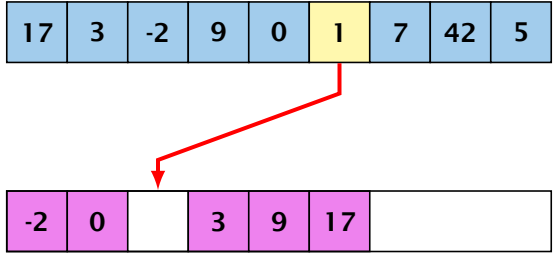
Beispiel



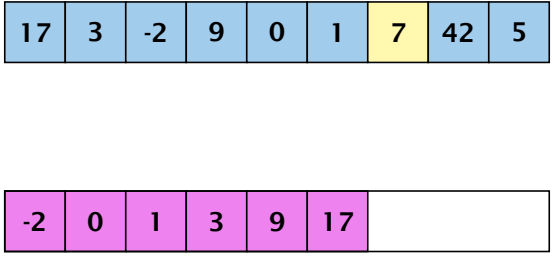
Beispiel



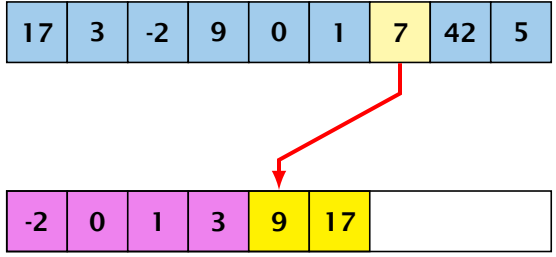
Beispiel



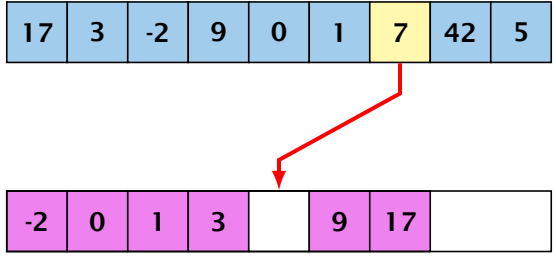
Beispiel



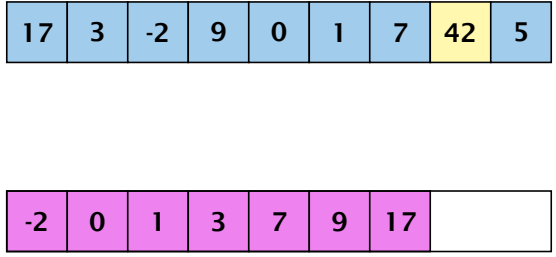
Beispiel



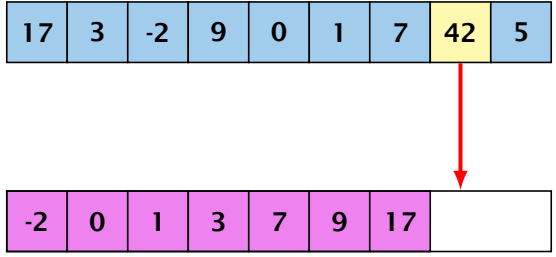
Beispiel



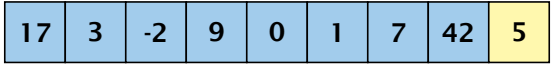
Beispiel



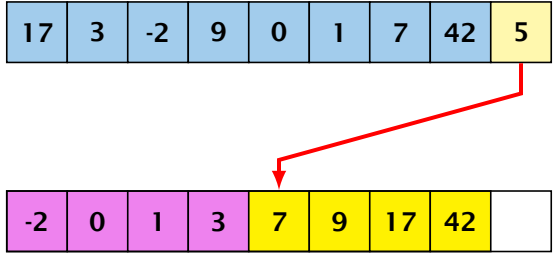
Beispiel



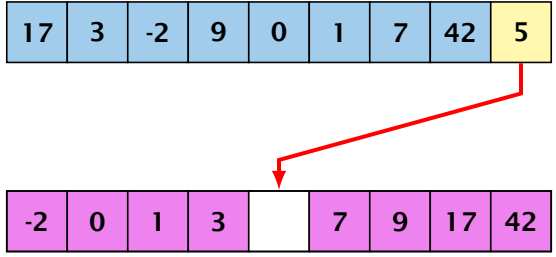
Beispiel



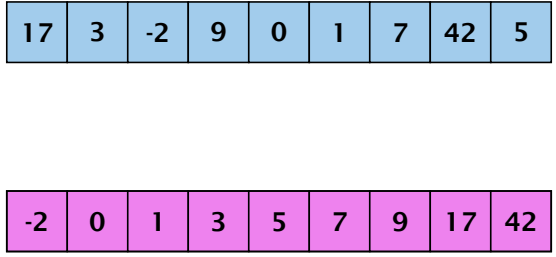
Beispiel



Beispiel



Beispiel



7 Anwendung: Sortieren

```
1 public static void insert(int[] b, int x, int i) {
2        // finde Einfuegestelle j fuer x in b
3        int j = locate(b,x,i);
4        // verschiebe in b Elemente b[j],...,b[i-1]
5        // nach rechts
6        shift(b,j,i);
7        b[j] = x;
8    }
```

Einfügen

- Wie findet man Einfügestelle?
- Wie verschiebt man nach rechts?

7 Anwendung: Sortieren

```
1 public static void insert(int[] b, int x, int i) {
2         // finde Einfuegestelle j fuer x in b
3         int j = locate(b,x,i);
4         // verschiebe in b Elemente b[j],...,b[i-1]
5         // nach rechts
6         shift(b,j,i);
7         b[j] = x;
8     }
```

Einfügen

- Wie findet man Einfügestelle?
- Wie verschiebt man nach rechts?

7 Anwendung: Sortieren

```
public static int locate(int[] b, int x, int i) {
    int j = 0;
    while (j < i && x > b[j]) ++j;
    return j;
}
public static void shift(int[] b, int j, int i) {
    for (int k = i-1; k >= j; --k)
        b[k+1] = b[k];
}
```

- Warum läuft Iteration in `shift()` von `i-1` abwärts nach `j`?

7 Anwendung: Sortieren

Erläuterungen

- Das Feld b ist (ursprünglich) lokale Variable von `sort()`.
- Lokale Variablen sind nur im eigenen Funktionsrumpf sichtbar, nicht in den aufgerufenen Funktionen.
- Damit die aufgerufenen Hilfsfunktionen auf b zugreifen können, muss b explizit als Parameter übergeben werden!

  Achtung:
  Das Feld wird nicht kopiert. Das Argument ist der Wert der Variablen b, also nur eine Referenz!
- Deshalb benötigen weder `insert()`, noch `shift()` einen separaten Rückgabewert...
- Weil das Problem so klein ist, würde eine erfahrene Programmiererin hier keine Unterprogramme benutzen...

7 Anwendung: Sortieren

```
public static int locate(int[] b, int x, int i) {
    int j = 0;
    while (j < i && x > b[j]) ++j;
    return j;
}
public static void shift(int[] b, int j, int i) {
    for (int k = i-1; k >= j; --k)
        b[k+1] = b[k];
}
```

- Warum läuft Iteration in `shift()` von `i-1` abwärts nach `j`?

7 Anwendung: Sortieren

```
1  public static int[] sort(int[] a) {
2      int[] b = new int[a.length];
3      for (int i = 0; i < a.length; ++i) {
4          // begin of insert
5          int j = 0;
6          while (j < i && a[i] > b[j]) ++j;
7          // end of locate
8          for (int k = i-1; k >= j; --k)
9              b[k+1] = b[k];
10         // end of shift
11         b[j] = a[i];
12         // end of insert
13     }
14     return b;
15 } // end of sort
```


7 Anwendung: Sortieren

Erläuterungen

- Das Feld b ist (ursprünglich) lokale Variable von `sort()`.
- Lokale Variablen sind nur im eigenen Funktionsrumpf sichtbar, nicht in den aufgerufenen Funktionen.
- Damit die aufgerufenen Hilfsfunktionen auf b zugreifen können, muss b explizit als Parameter übergeben werden!

  Achtung:
  Das Feld wird nicht kopiert. Das Argument ist der Wert der Variablen b, also nur eine Referenz!
- Deshalb benötigen weder `insert()`, noch `shift()` einen separaten Rückgabewert...
- Weil das Problem so klein ist, würde eine erfahrene Programmiererin hier keine Unterprogramme benutzen...

7 Anwendung: Sortieren

```
1  public static int[] sort(int[] a) {
2      int[] b = new int[a.length];
3      for (int i = 0; i < a.length; ++i) {
4          // begin of insert
5          int j = 0;
6          while (j < i && a[i] > b[j]) ++j;
7          // end of locate
8          for (int k = i-1; k >= j; --k)
9              b[k+1] = b[k];
10         // end of shift
11         b[j] = a[i];
12         // end of insert
13     }
14     return b;
15 } // end of sort
```


7 Anwendung: Sortieren

Diskussion

- Die Anzahl der ausgeführten Operationen wächst quadratisch in der Größe des Felds a.
- Glücklicherweise gibt es Sortierverfahren, die eine bessere Laufzeit haben (↑Algorithmen und Datenstrukturen).

8 Anwendung: Suchen

Gegeben: Folge `a` ganzer Zahlen; Element `x`

Gesucht: Wo kommt `x` in `a` vor?

Naives Vorgehen:

- Vergleiche `x` der Reihe nach mit `a[0]`, `a[1]`, usw.
- Finden wir `i` mit `a[i] == x`, geben wir `i` aus.
- Andernfalls geben wir `-1` aus: „Element nicht gefunden“!

8 Anwendung: Suchen

Gegeben: Folge a ganzer Zahlen; Element x

Gesucht: Wo kommt x in a vor?

Naives Vorgehen:

- Vergleiche x der Reihe nach mit a[0], a[1], usw.
- Finden wir i mit a[i] == x, geben wir i aus.
- Andernfalls geben wir -1 aus: „Element nicht gefunden“!

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

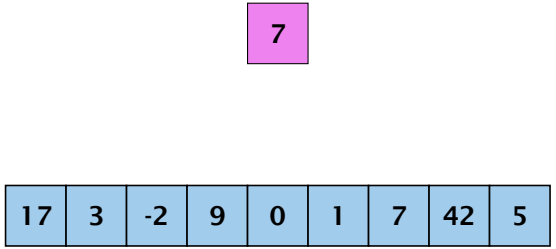
Naives Suchen

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

Naives Suchen

Beispiel

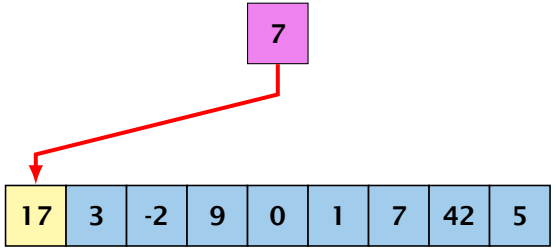


Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

Naives Suchen

Beispiel



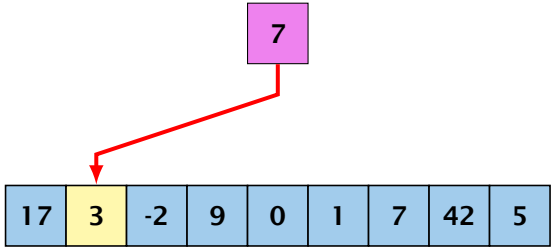
no

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

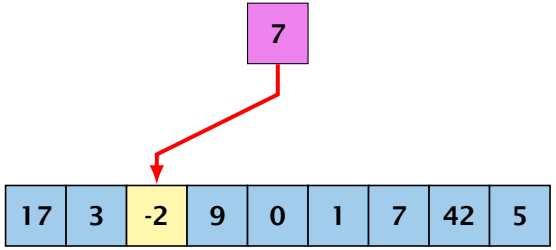
Naives Suchen

Beispiel



no

Beispiel



no

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

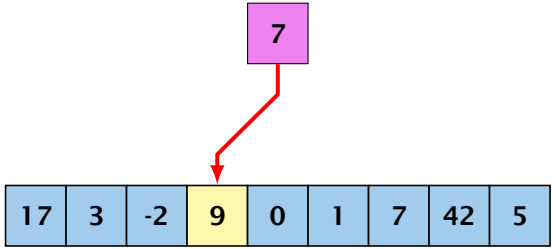
Naives Suchen

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

Naives Suchen

Beispiel



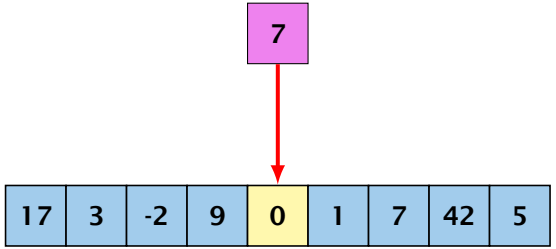
no

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

Naives Suchen

Beispiel



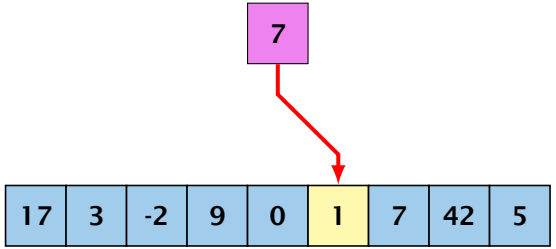
no

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

Naives Suchen

Beispiel



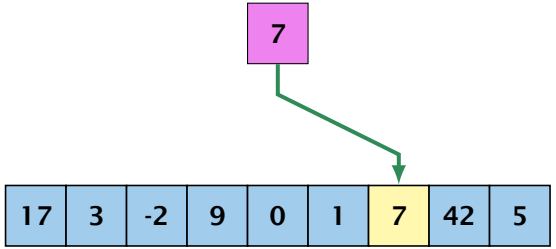
no

Naives Suchen

```
1 public static int find(int[] a, int x) {
2     int i = 0;
3     while (i < a.length && a[i] != x)
4         ++i;
5     if (i == a.length)
6         return -1;
7     else
8         return i;
9 }
```

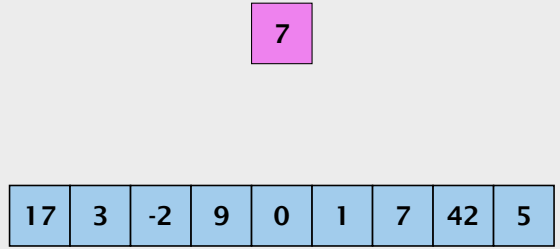
Naives Suchen

Beispiel



yes

Beispiel



no

Naives Suchen

- Im Beispiel benötigen wir 7 Vergleiche
- Im schlimmsten Fall (worst case) benötigen wir bei einem Feld der Länge n sogar n Vergleiche.
- Kommt x tatsächlich im Feld vor, benötigen wir selbst im Durchschnitt $(n+1)/2$ Vergleiche.

...geht das nicht besser?

Naives Suchen

- Im Beispiel benötigen wir 7 Vergleiche
- Im schlimmsten Fall (worst case) benötigen wir bei einem Feld der Länge n sogar n Vergleiche.
- Kommt x tatsächlich im Feld vor, benötigen wir selbst im Durchschnitt $(n+1)/2$ Vergleiche.

...geht das nicht besser?

Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

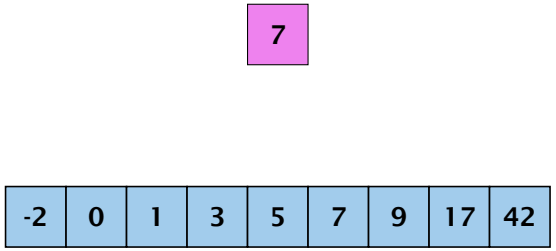
Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

Beispiel



- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

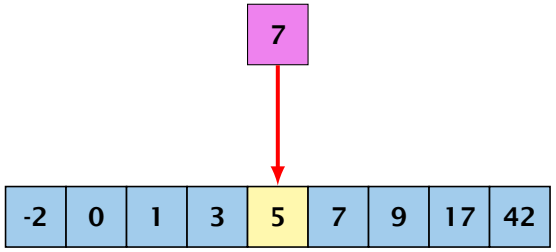
Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

Beispiel



no

- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

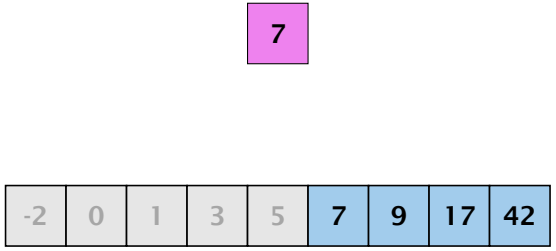
Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

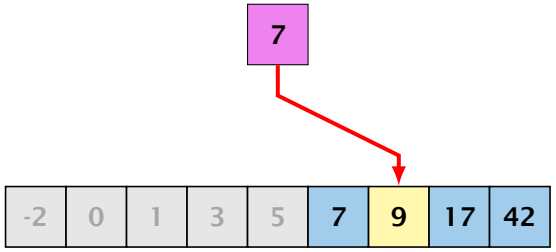
$\Longrightarrow$ binäre Suche

Beispiel



- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel



no

- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

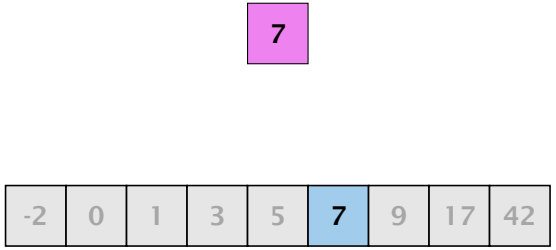
Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

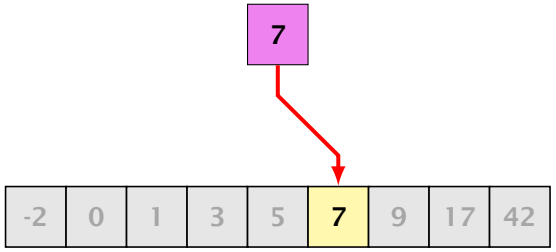
$\Longrightarrow$ binäre Suche

Beispiel



- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel



yes

- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

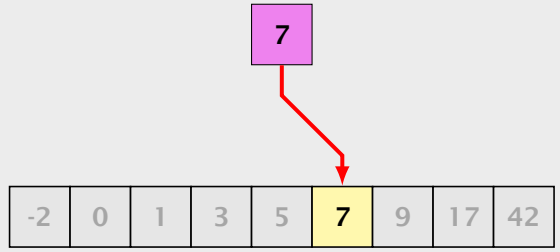
Binäre Suche

Idee:

- Sortiere das Feld.
- Vergleiche x mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist x kleiner, brauchen wir nur noch links weitersuchen.
- Ist x größer, brauchen wir nur noch rechts weiter suchen.

$\Longrightarrow$ binäre Suche

Beispiel



yes

- wir benötigen nur drei Vergleiche
- hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Implementierung

Idee:

Führe Hilfsfunktion

```
public static int find0(int[] a,int x,int n1,int n2)
```

ein, die im Interval [n1, n2] sucht.

Damit:

```
public static int find(int[] a, int x) {
    return find0(a,x,0,a.length-1);
}
```

Implementierung

Idee:

Führe Hilfsfunktion

```
public static int find0(int[] a,int x,int n1,int n2)
```

ein, die im Interval $[n1, n2]$ sucht.

Damit:

```
public static int find(int[] a, int x) {
    return find0(a,x,0,a.length-1);
}
```

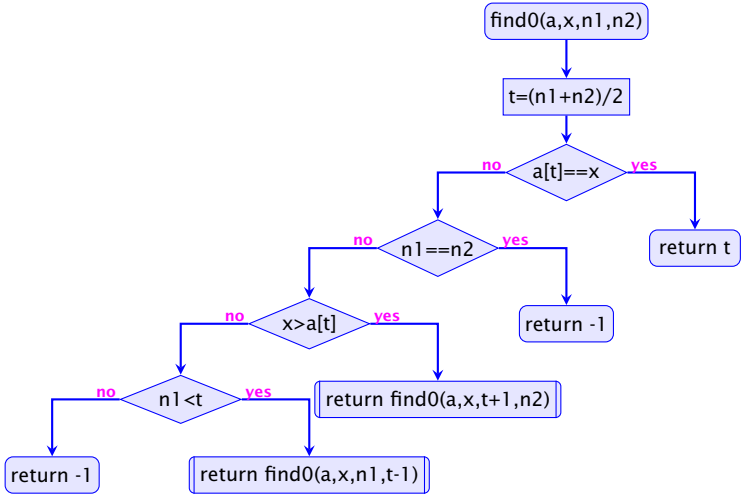

Implementierung

```
1  public static int find0(int[] a, int x, int n1, int n2) {
2      int t = (n1 + n2) / 2;
3      if (a[t] == x)
4          return t;
5      else if (n1 == n2)
6          return -1;
7      else if (x > a[t])
8          return find0(a, x, t+1, n2);
9      else if (n1 < t)
10         return find0(a, x, n1, t-1);
11     else return -1;
12 }
```

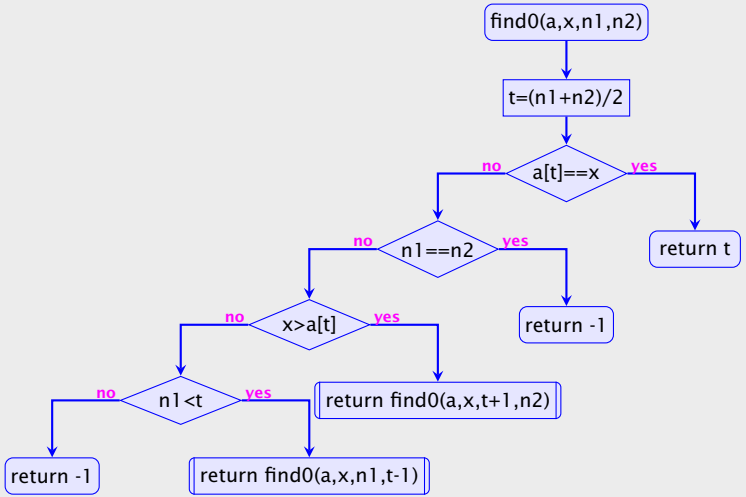
Implementierung

```
1  public static int find0(int[] a, int x, int n1, int n2) {
2      int t = (n1 + n2) / 2;
3      if (a[t] == x)
4          return t;
5      else if (n1 == n2)
6          return -1;
7      else if (x > a[t])
8          return find0(a, x, t+1, n2);
9      else if (n1 < t)
10         return find0(a, x, n1, t-1);
11     else return -1;
12 }
```


Kontrollflussdiagramm für find0



Kontrollflussdiagramm für `find0`

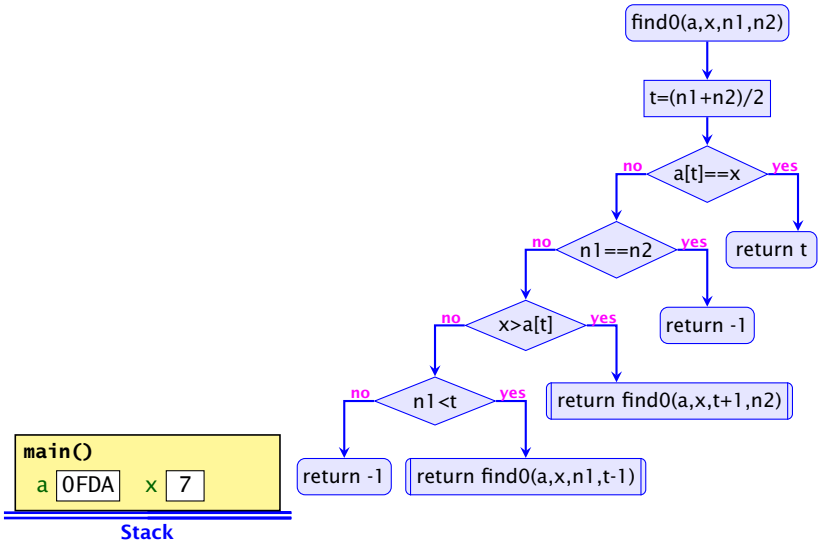


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

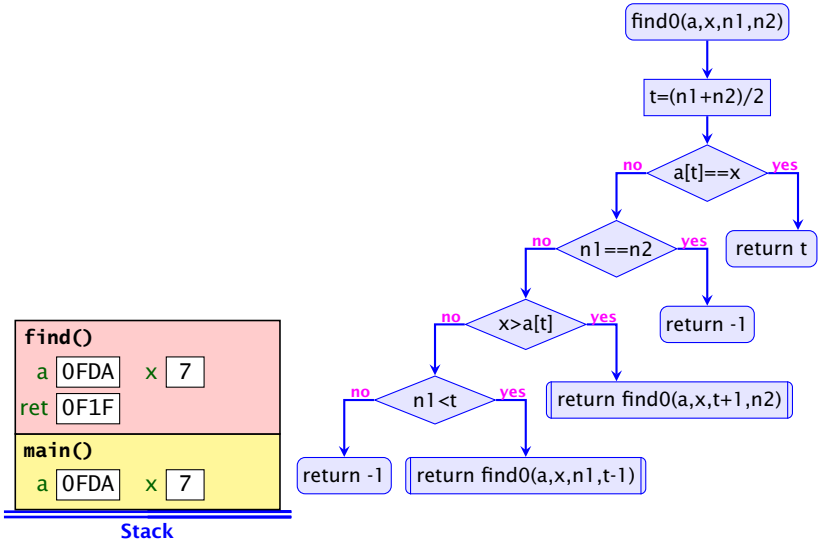


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

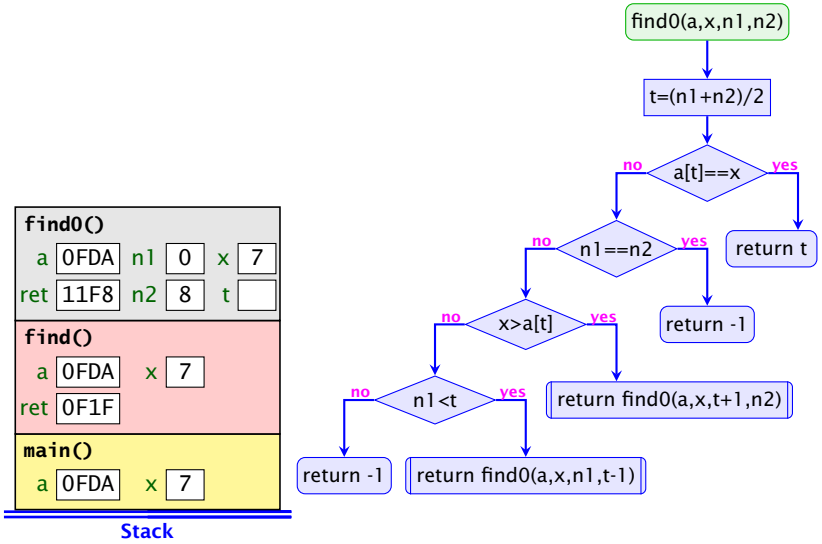


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

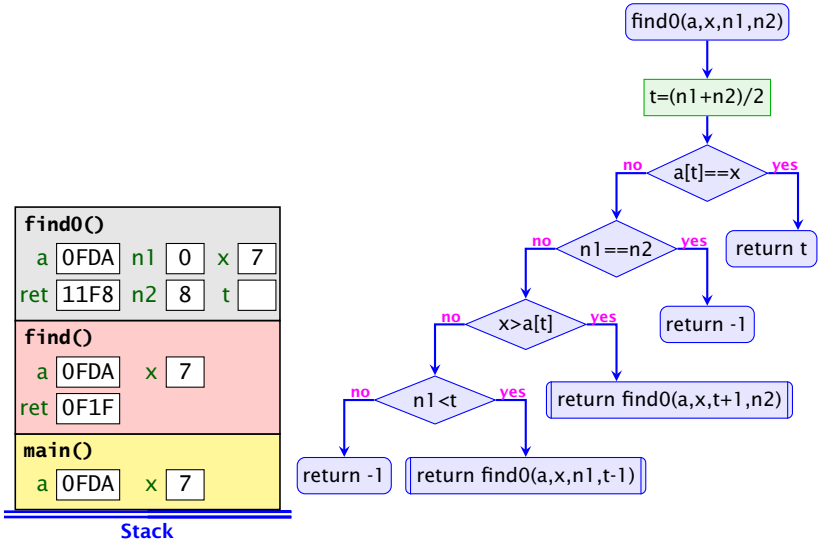


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

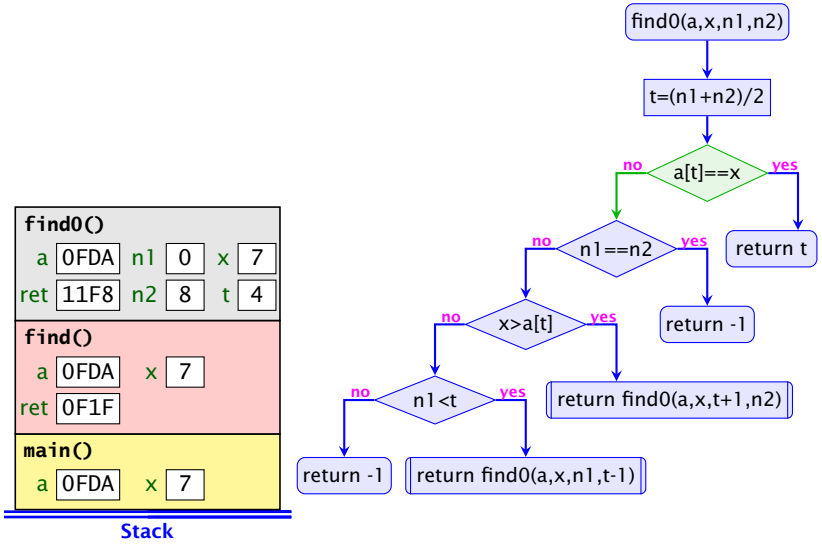


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

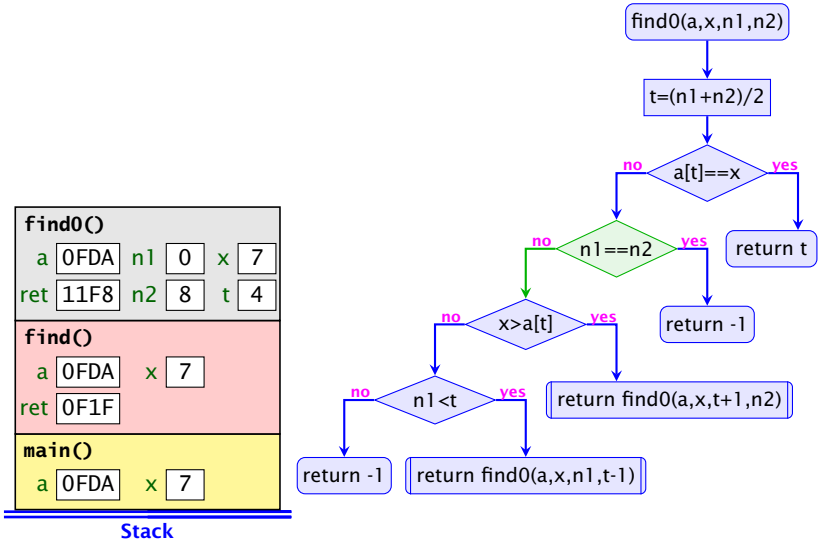


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

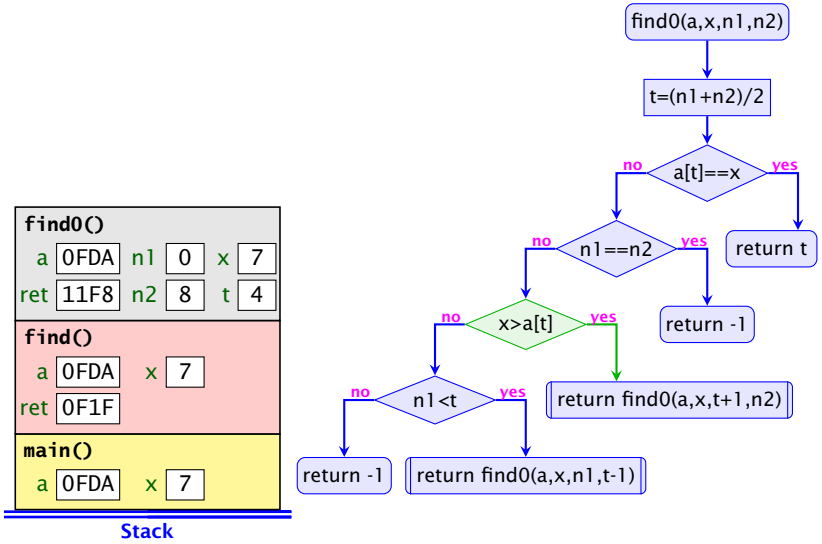


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

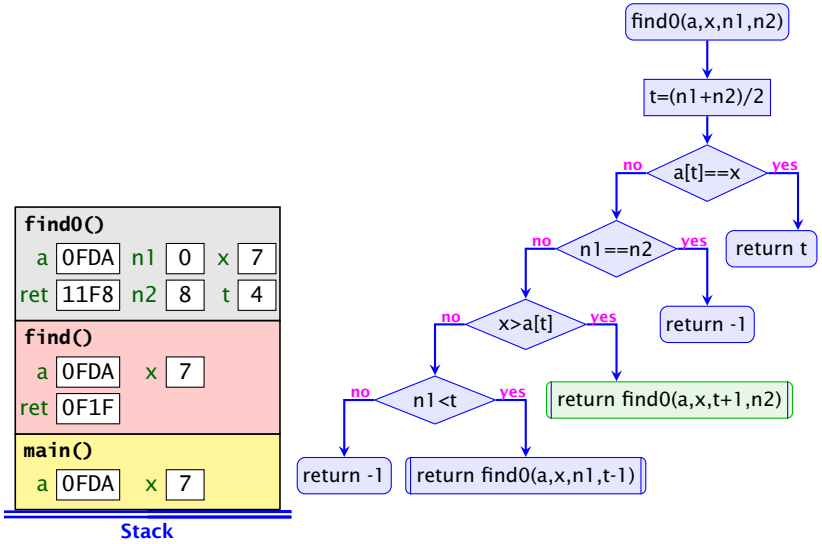


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

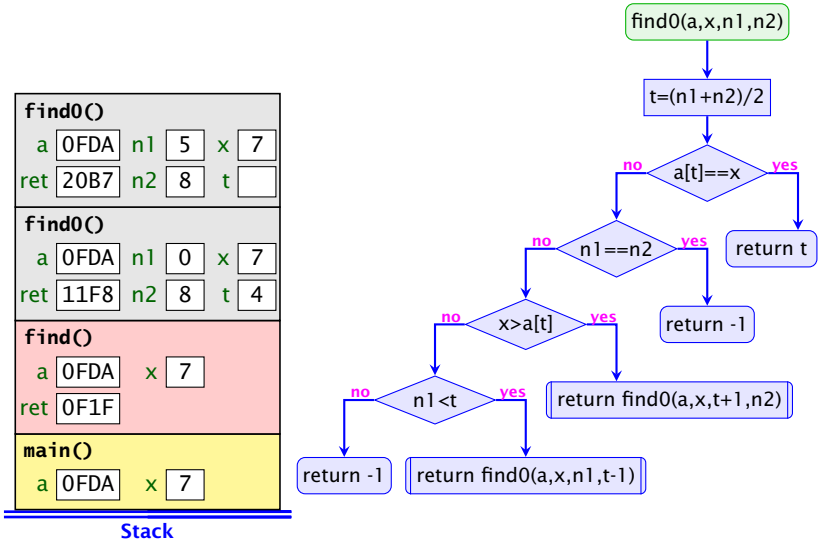


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

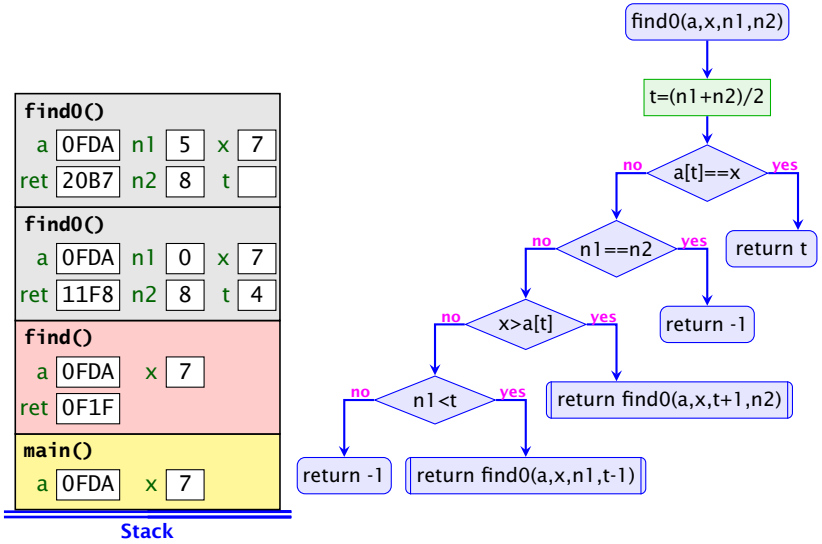


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

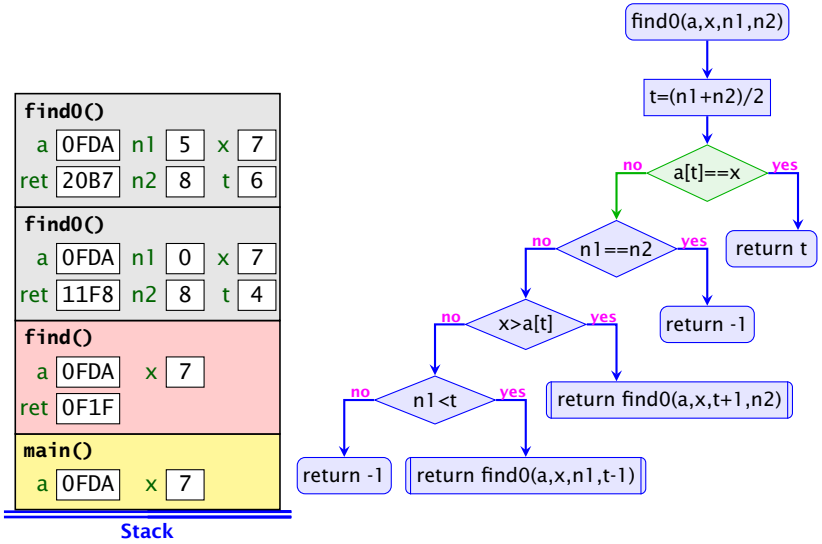


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

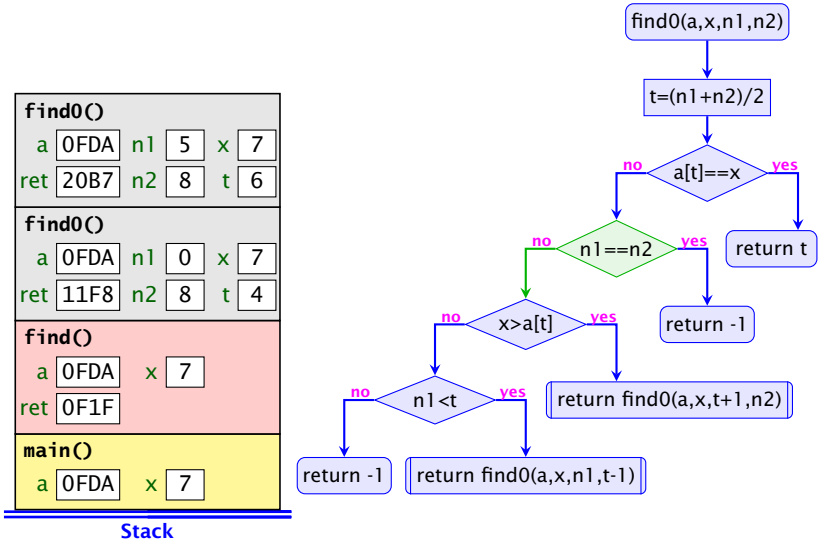


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

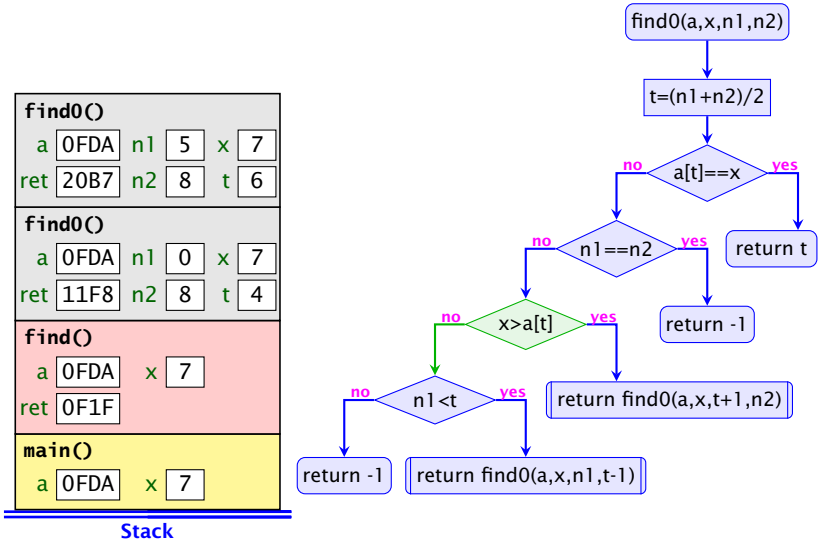


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

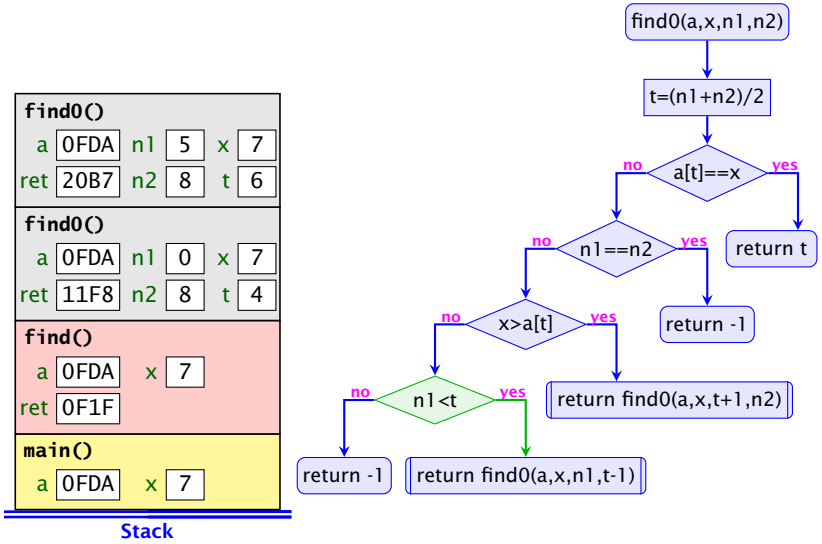


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

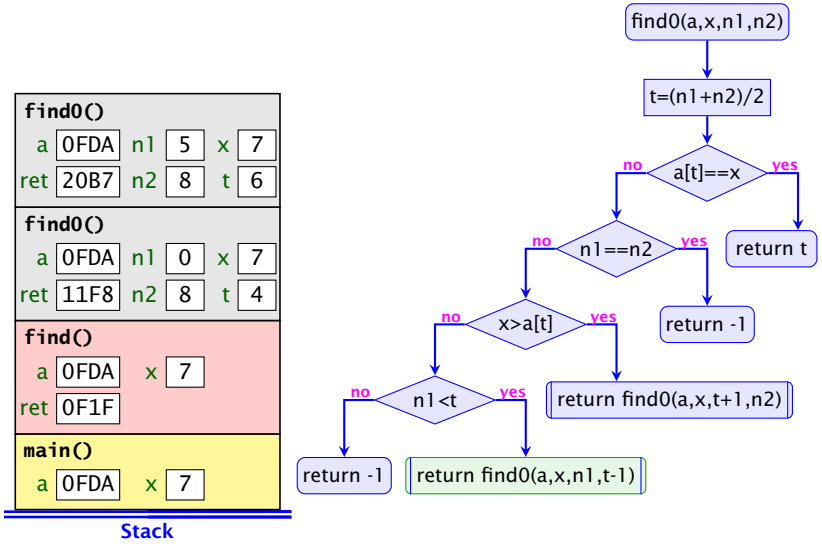


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

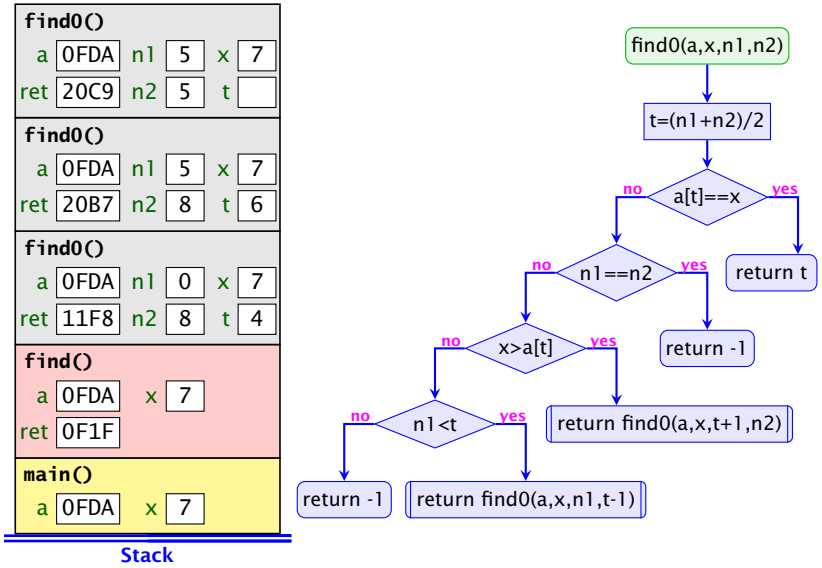


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

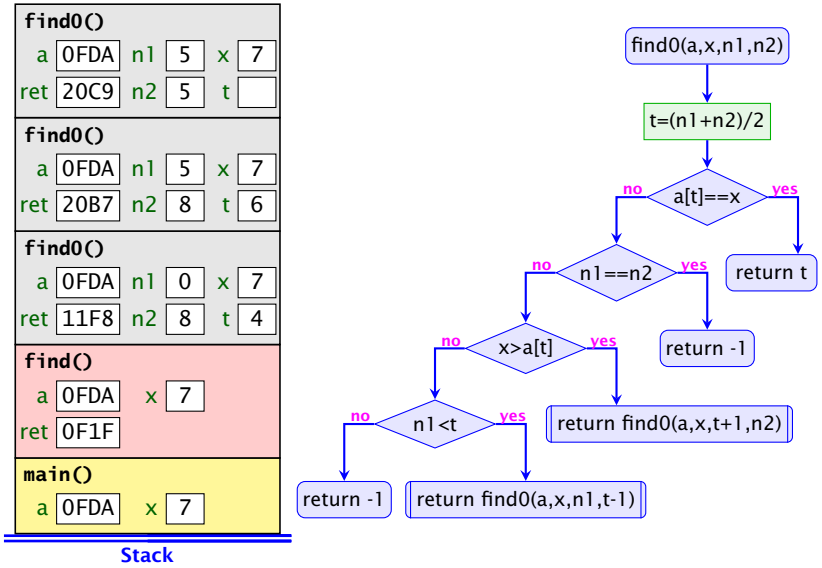


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

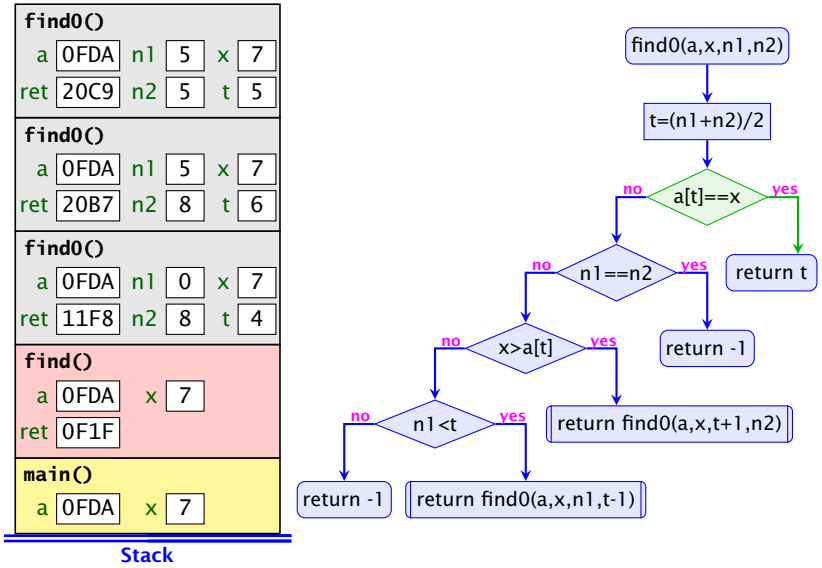


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

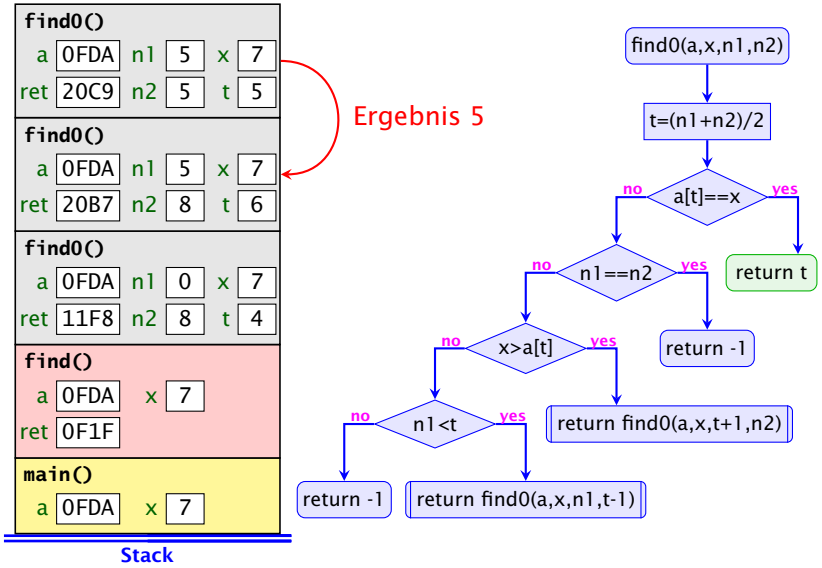


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

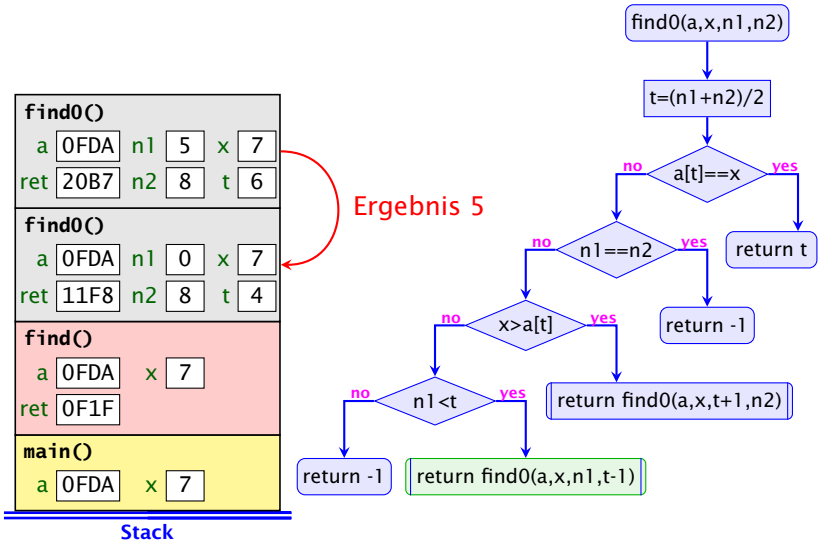


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

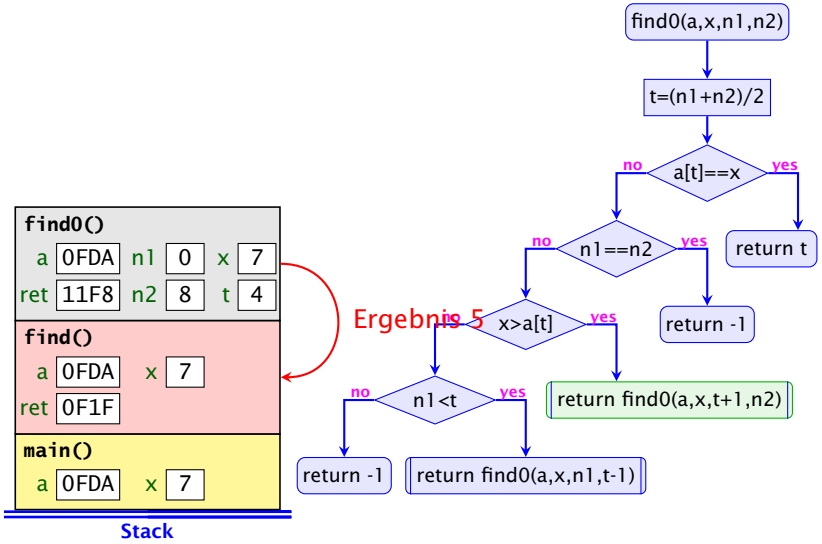


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung

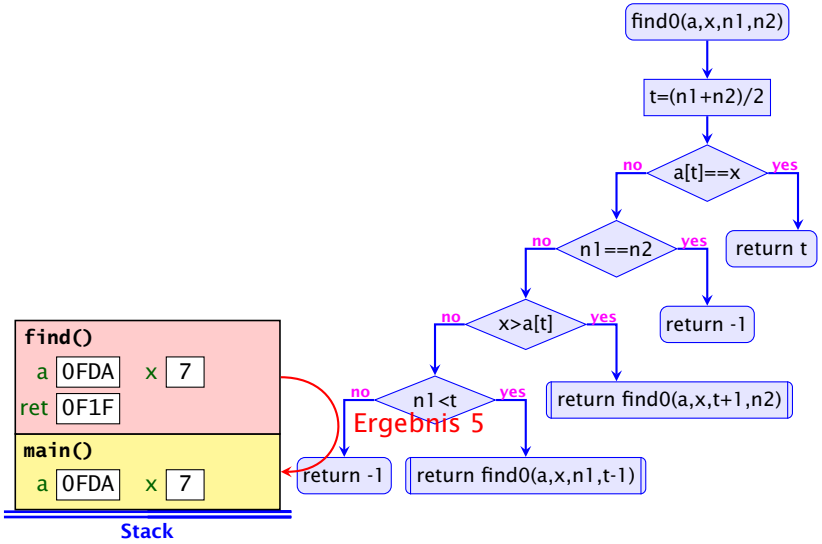


Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

Ausführung



Implementierung

Erläuterungen:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen rekursiv.

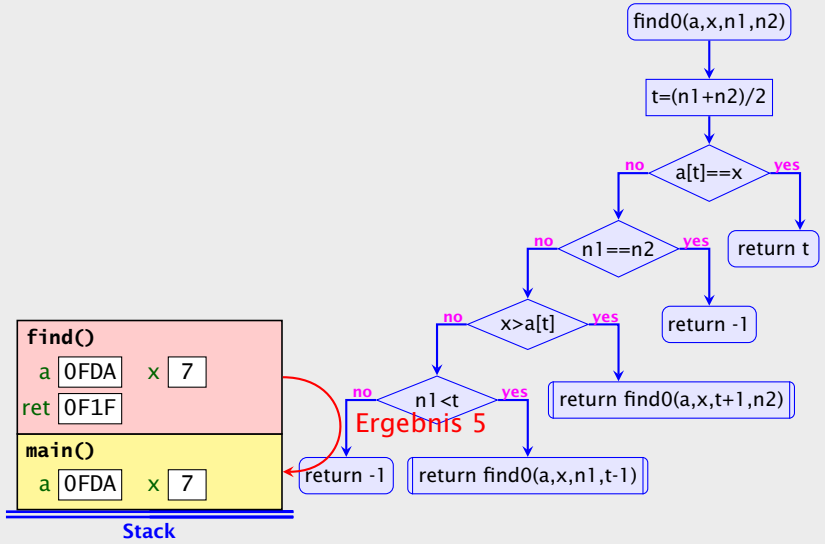
Terminierung

Um zu beweisen, dass `find0()` terminiert, beobachten wir:

1. Wird `find0()` für ein einelementiges Intervall `[n,n]` aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `find0()` für ein Intervall `[n1,n2]` aufgerufen mit mehr als einem Element, dann terminiert der Aufruf entweder direkt (weil `x` gefunden wurde), oder `find0()` wird mit einem Intervall aufgerufen, das echt in `[n1,n2]` enthalten ist, genauer: sogar maximal die Hälfte der Elemente von `[n1,n2]` enthält.

Ähnliche Beweistechnik wird auch für andere rekursive Funktionen verwendet.

Ausführung



Terminierung

Um zu beweisen, dass `findO()` terminiert, beobachten wir:

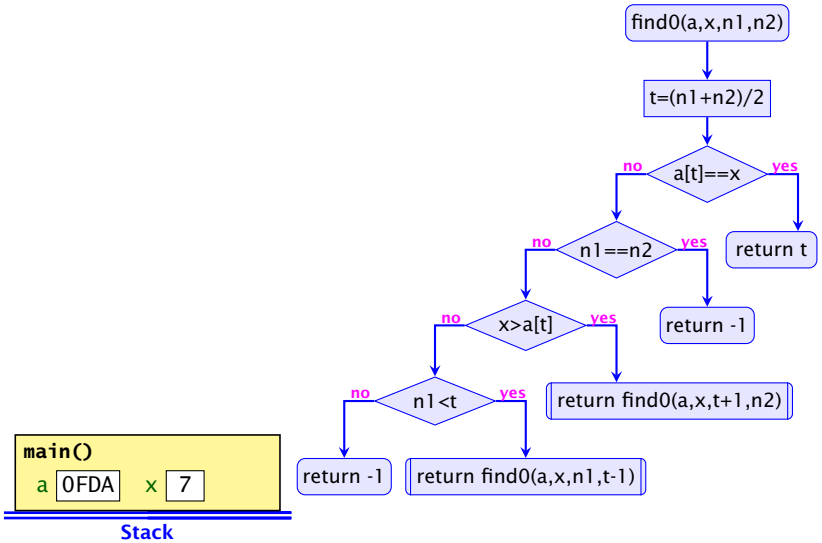
1. Wird `findO()` für ein einelementiges Intervall `[n,n]` aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `findO()` für ein Intervall `[n1,n2]` aufgerufen mit mehr als einem Element, dann terminiert der Aufruf entweder direkt (weil `x` gefunden wurde), oder `findO()` wird mit einem Intervall aufgerufen, das echt in `[n1,n2]` enthalten ist, genauer: sogar maximal die Hälfte der Elemente von `[n1,n2]` enthält.

Ähnliche Beweistechnik wird auch für andere rekursive Funktionen verwendet.

Beobachtung

- Das Ergebnis eines Aufrufs von `findO()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `findO()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

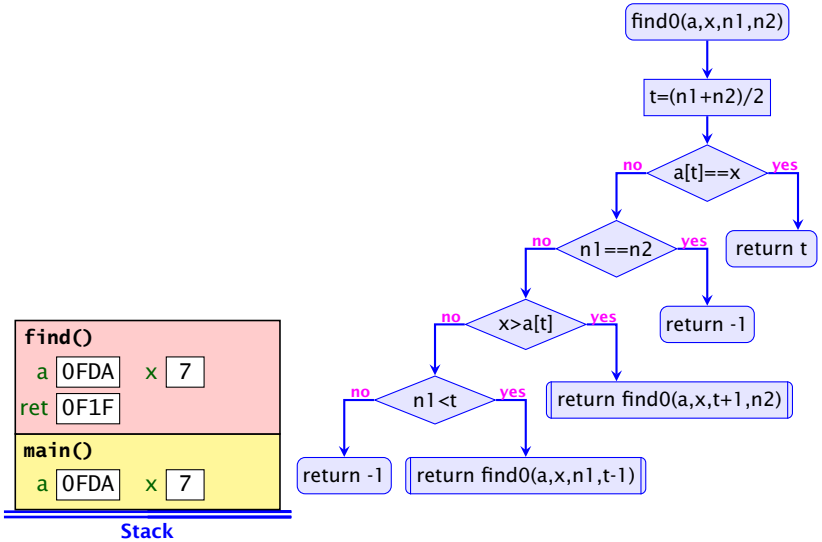
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

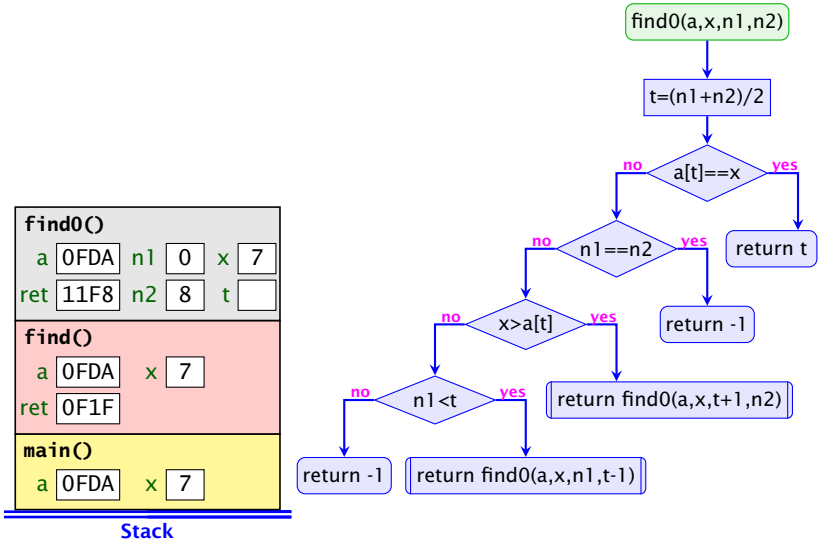
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

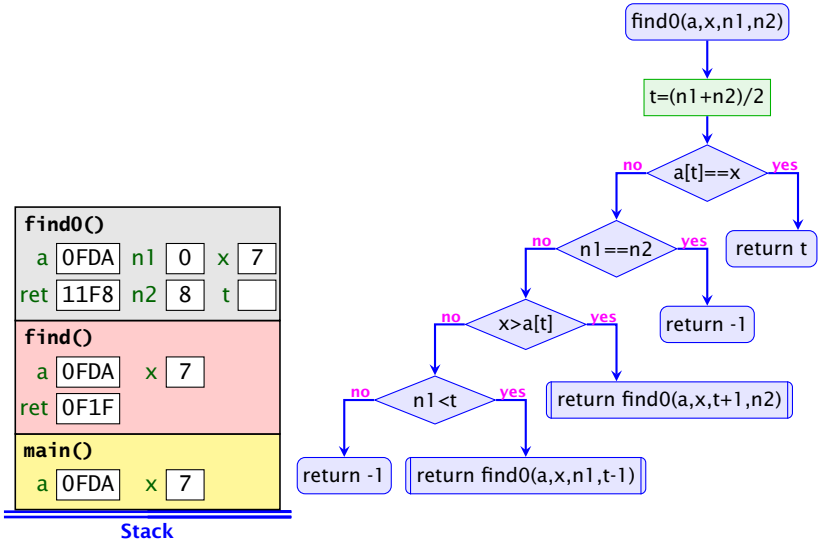
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

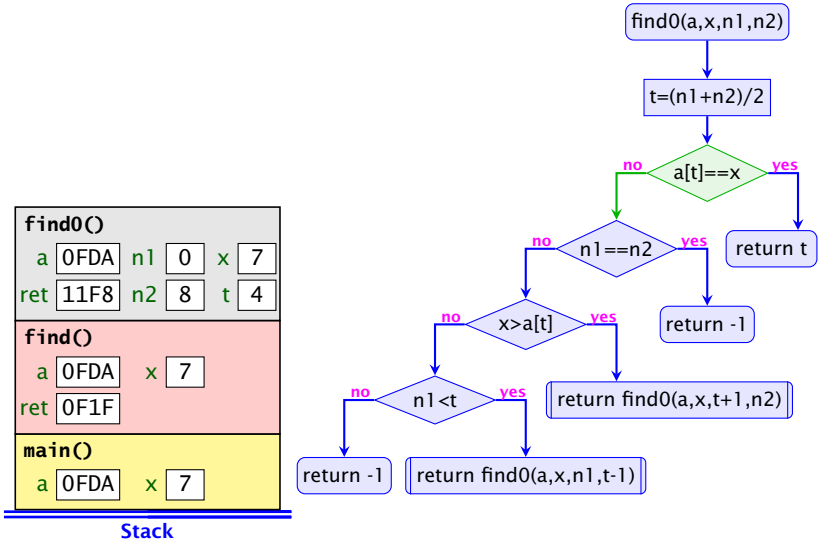
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

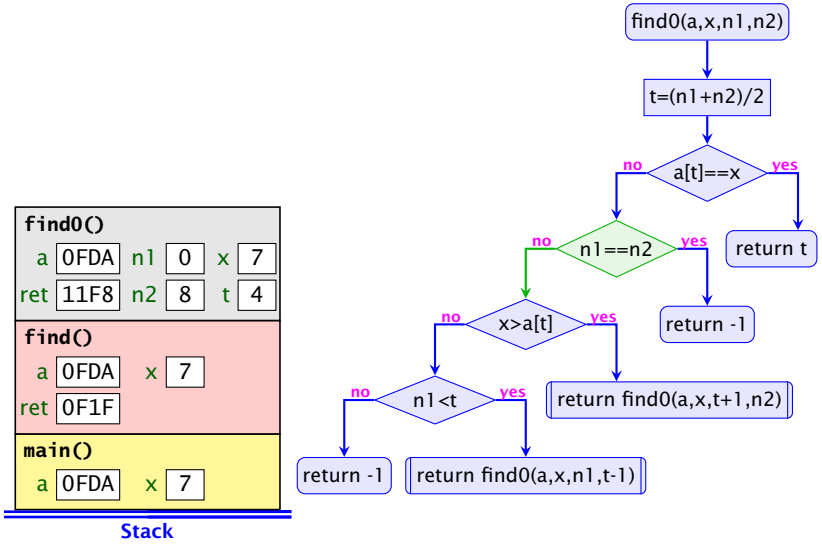
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

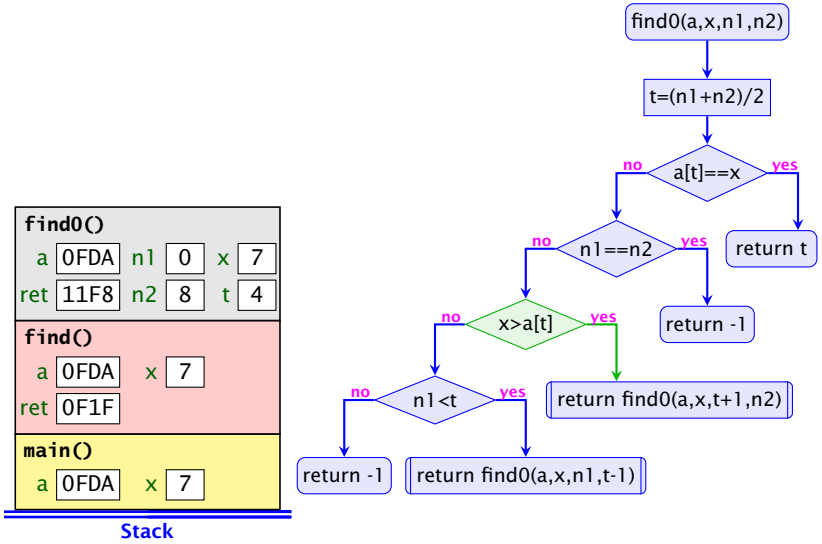
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

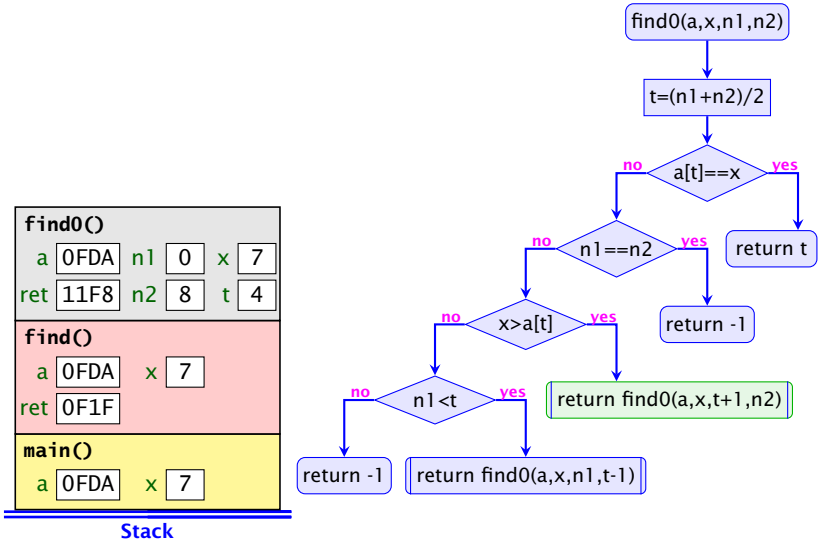
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

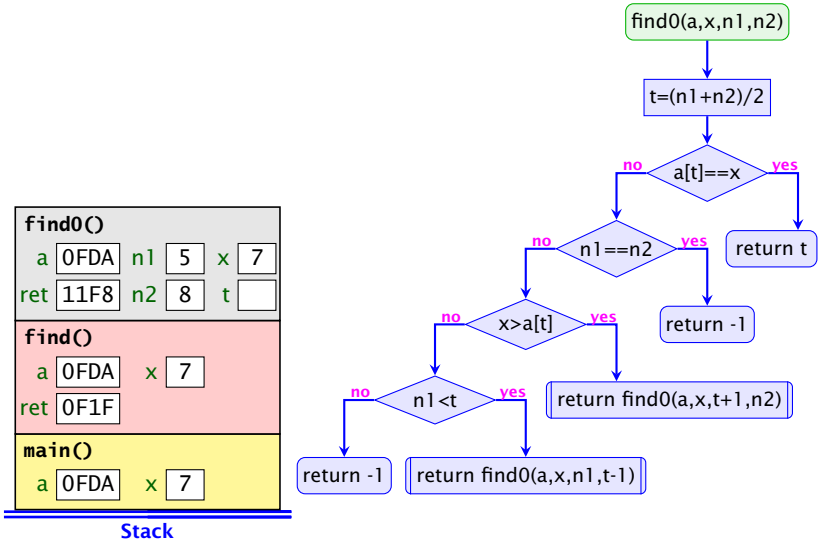
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

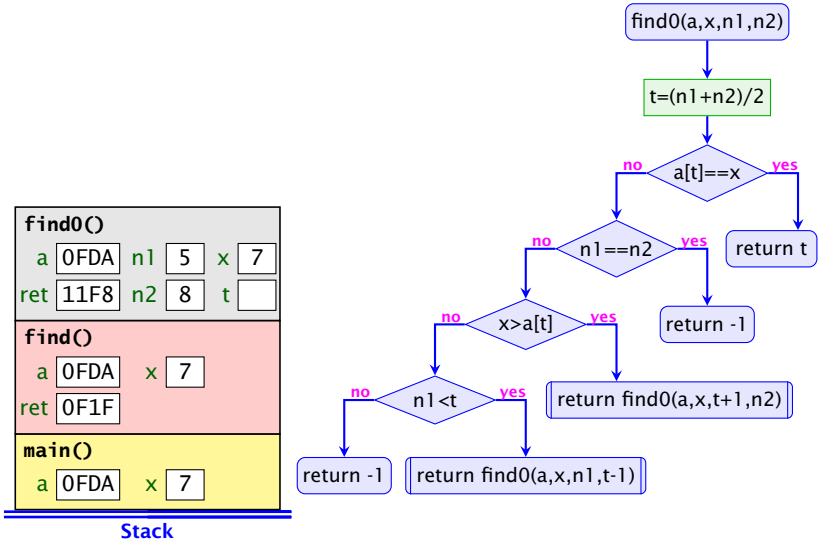
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

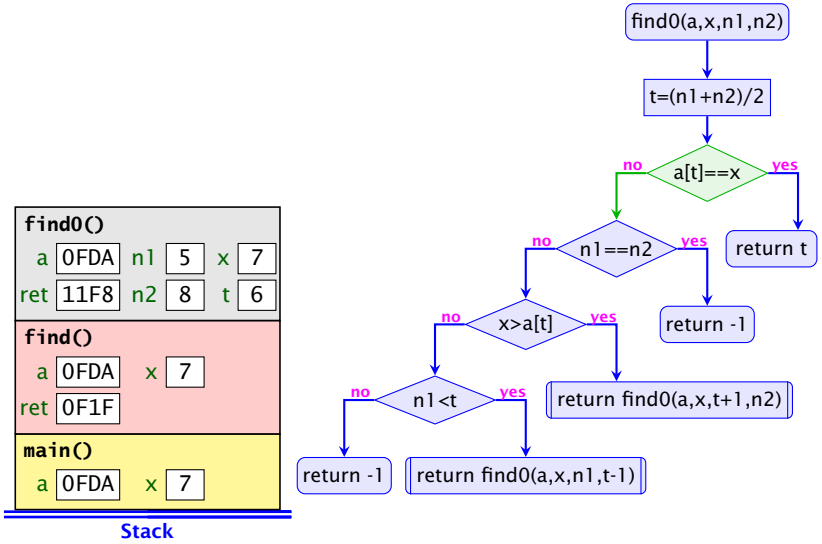
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

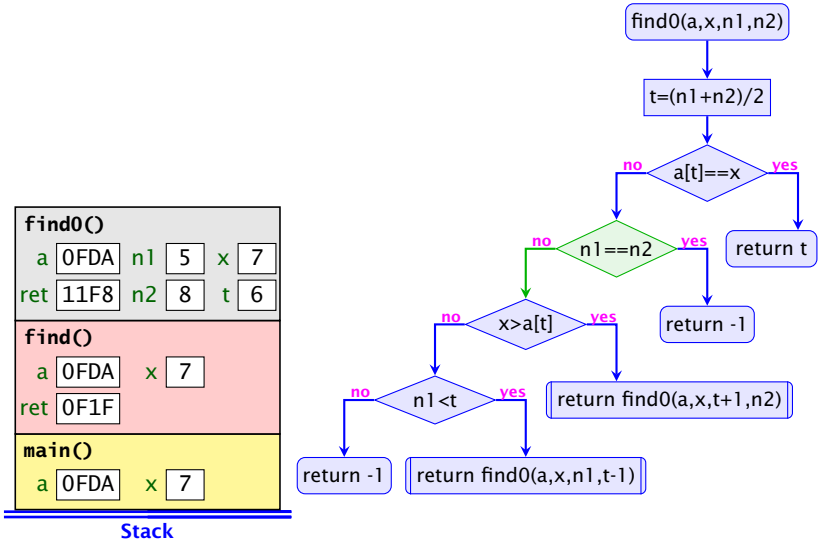
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

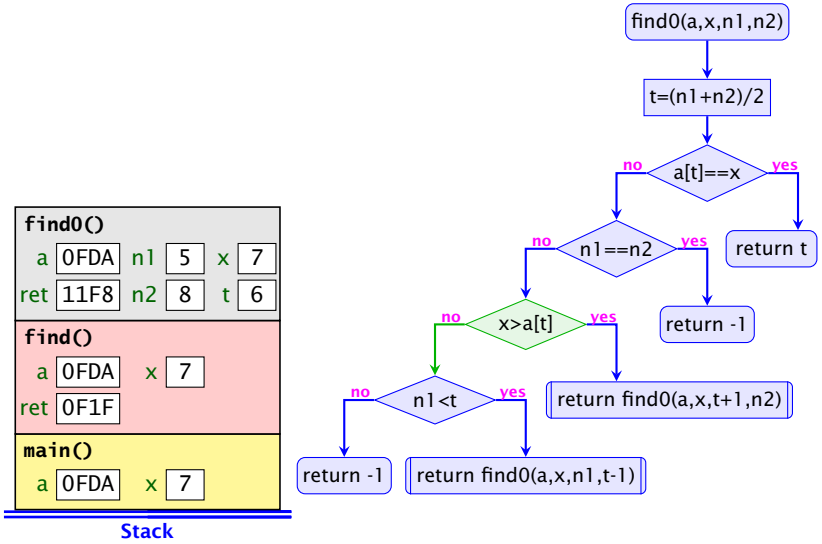
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

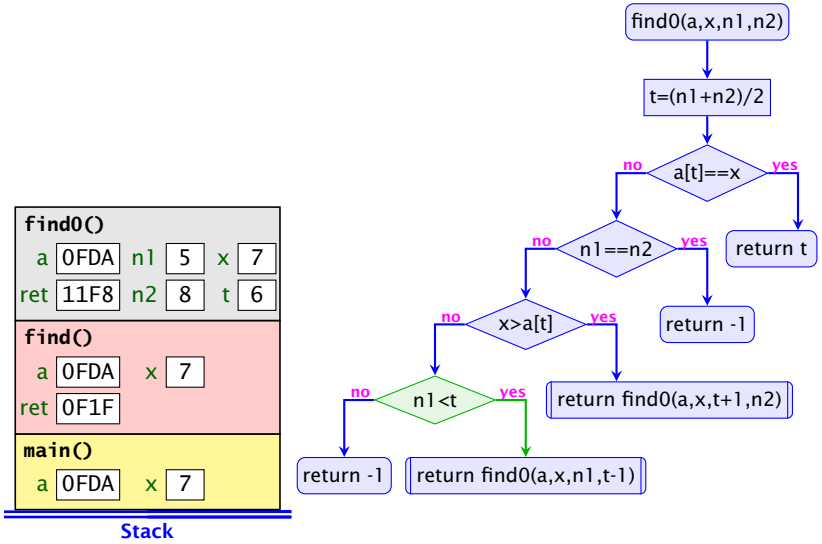
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

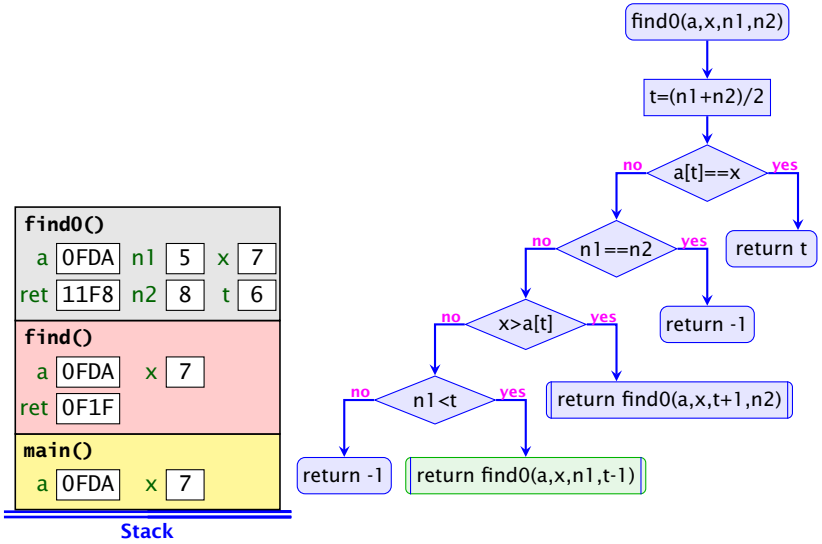
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

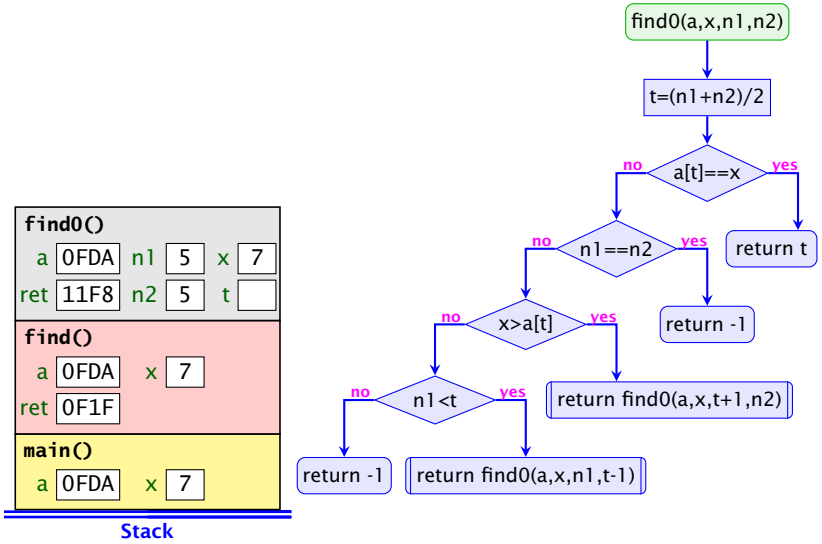
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

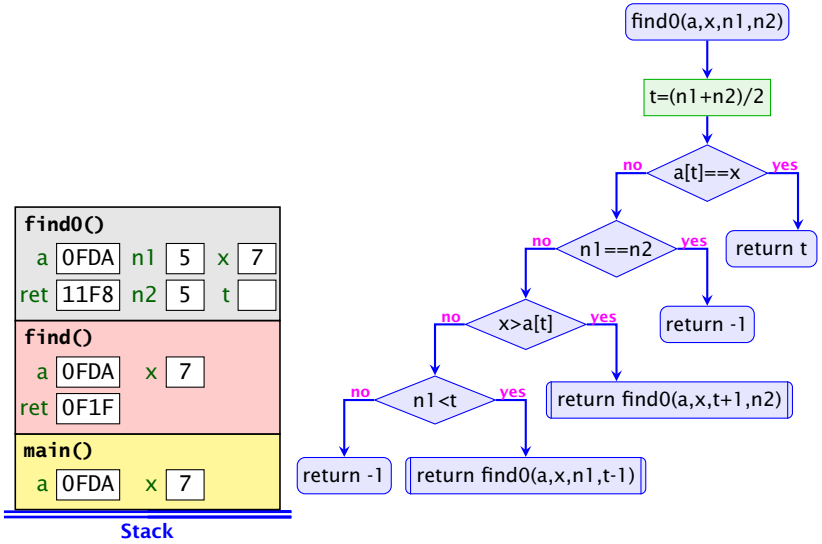
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

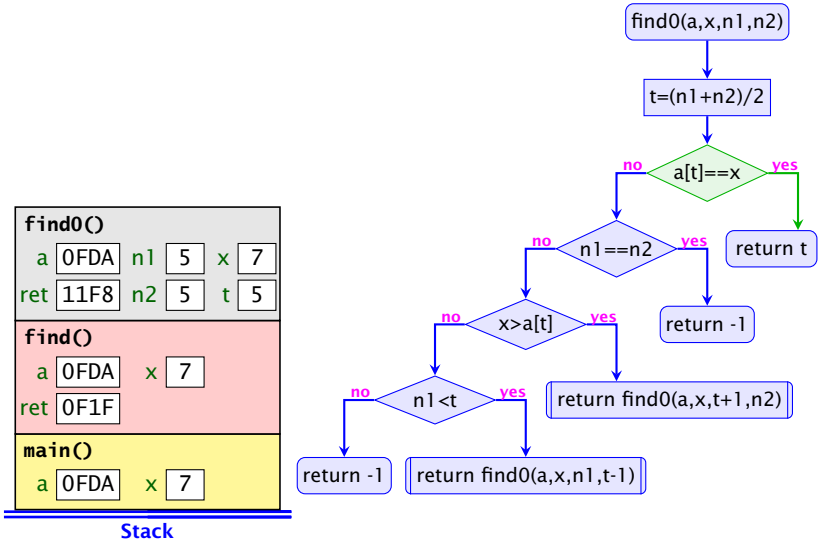
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

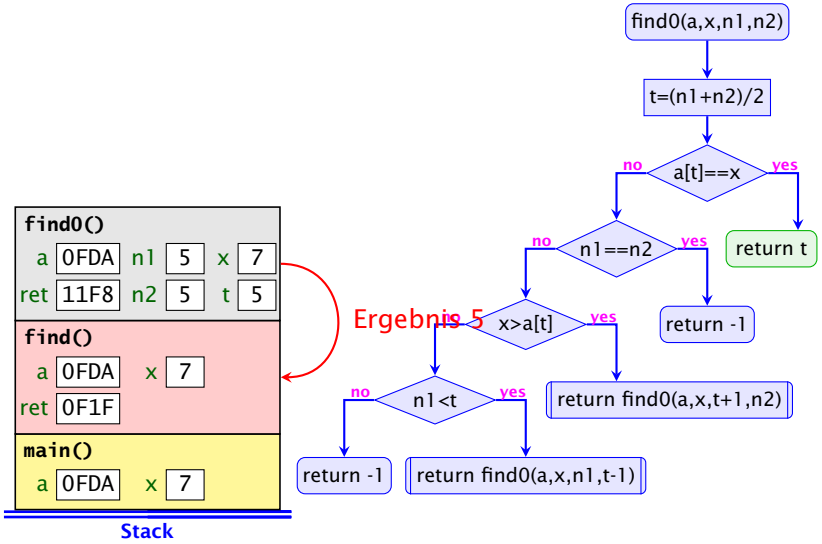
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

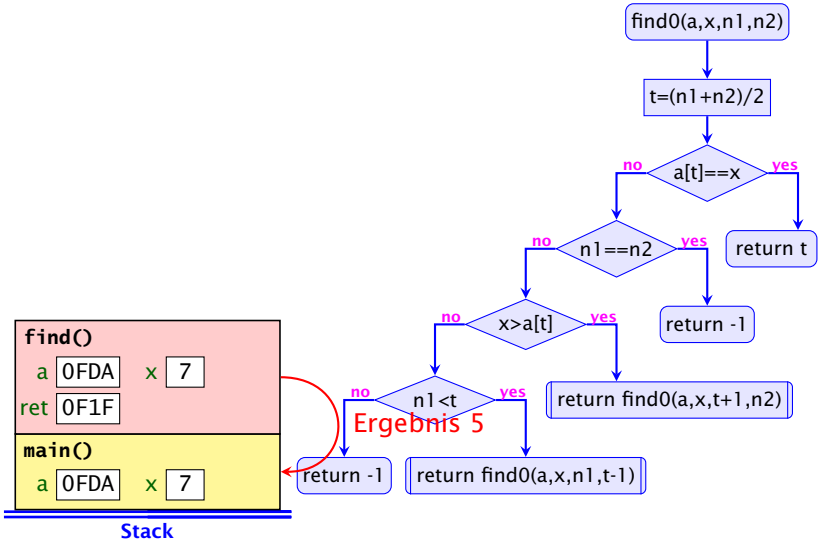
Verbesserte Ausführung



Beobachtung

- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

Verbesserte Ausführung



Beobachtung

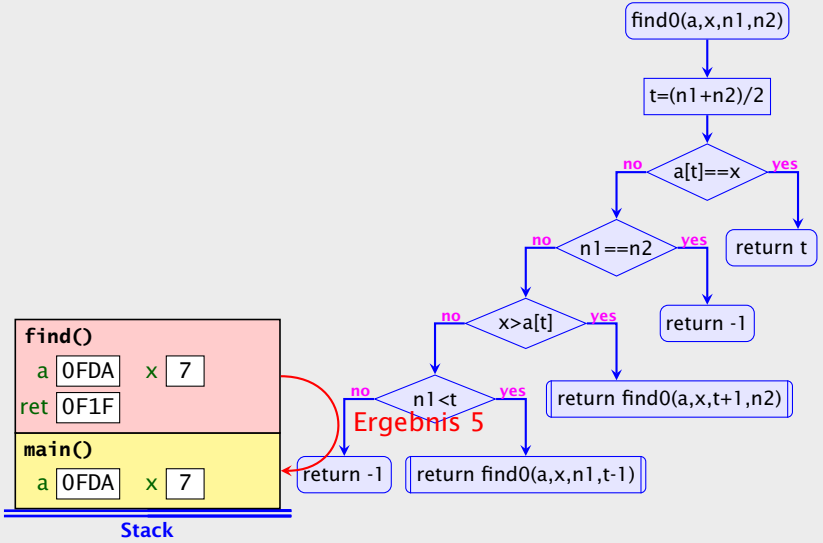
- Das Ergebnis eines Aufrufs von `find0()` liefert direkt das Ergebnis auch für die aufrufende Funktion!
- Solche Rekursion heißt End- oder Tail-Rekursion.
- End-Rekursion kann auch ohne Aufrufkeller implementiert werden...
- **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern ersetze den bereits dort liegenden Aufruf!

Endrekursion

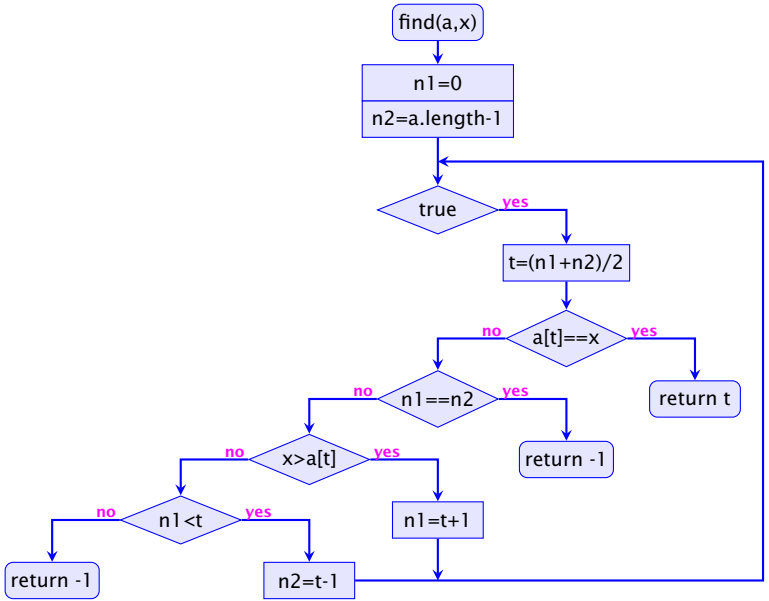
Endrekursion kann durch Iteration ersetzt werden...

```
1  public static int find(int[] a, int x) {
2      int n1 = 0;
3      int n2 = a.length-1;
4      while (true) {
5          int t = (n2 + n1) / 2;
6          if (x == a[t]) return t;
7          else if (n1 == n2) return -1;
8          else if (x > a[t]) n1 = t+1;
9          else if (n1 < t) n2 = t-1;
10         else return -1;
11     } // end of while
12 } // end of find
```

Verbesserte Ausführung



Kontrollfluss



Endrekursion

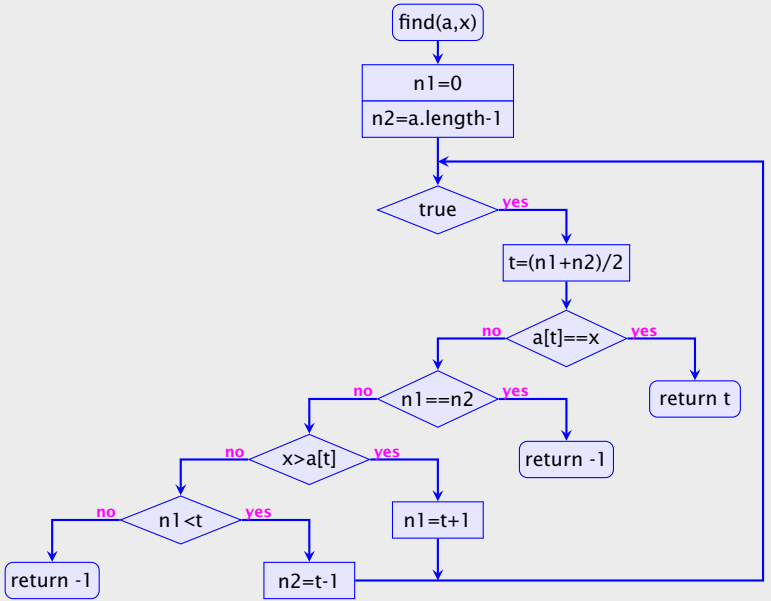
Endrekursion kann durch Iteration ersetzt werden...

```
1  public static int find(int[] a, int x) {
2      int n1 = 0;
3      int n2 = a.length-1;
4      while (true) {
5          int t = (n2 + n1) / 2;
6          if (x == a[t]) return t;
7          else if (n1 == n2) return -1;
8          else if (x > a[t]) n1 = t+1;
9          else if (n1 < t) n2 = t-1;
10         else return -1;
11     } // end of while
12 } // end of find
```

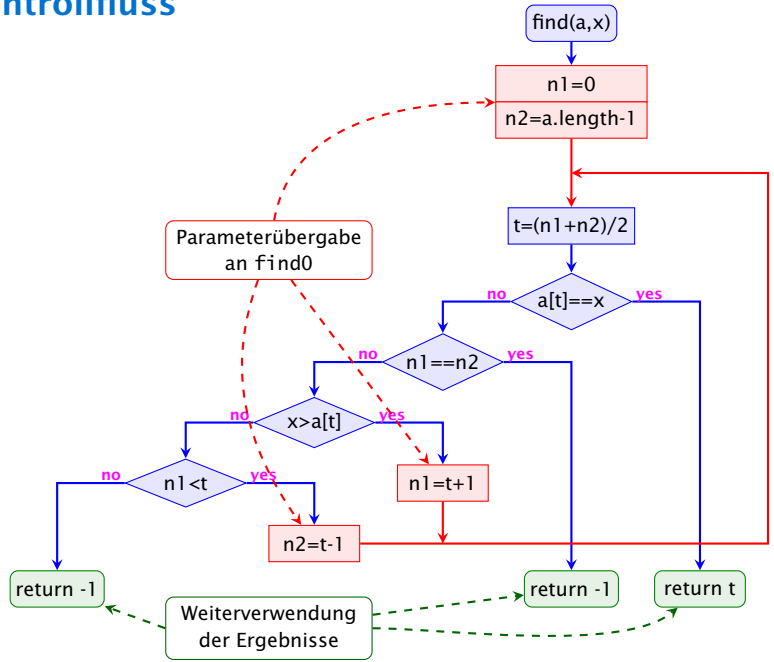
Verlassen von Schleifen

- Die Schleife wird hier alleine durch die `return`-Anweisungen verlassen.
- Offenbar machen Schleifen mit mehreren Ausgängen Sinn.
- Um eine Schleife zu verlassen, ohne gleich ans Ende der Funktion zu springen, kann man das `break`-Statement benutzen.
- Der Aufruf der endrekursiven Funktion wird ersetzt durch:
  1. Code zur Parameter-Übergabe;
  2. einen Sprung an den Anfang des Rumpfs.

Kontrollfluss



Kontrollfluss



Verlassen von Schleifen

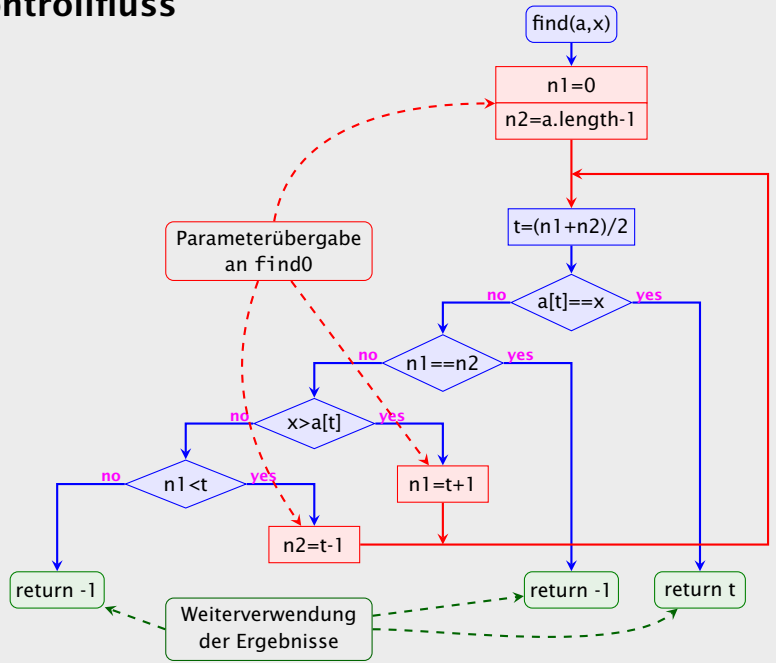
- Die Schleife wird hier alleine durch die `return`-Anweisungen verlassen.
- Offenbar machen Schleifen mit mehreren Ausgängen Sinn.
- Um eine Schleife zu verlassen, ohne gleich ans Ende der Funktion zu springen, kann man das `break`-Statement benutzen.
- Der Aufruf der endrekursiven Funktion wird ersetzt durch:
  1. Code zur Parameter-Übergabe;
  2. einen Sprung an den Anfang des Rumpfs.

Rekursion

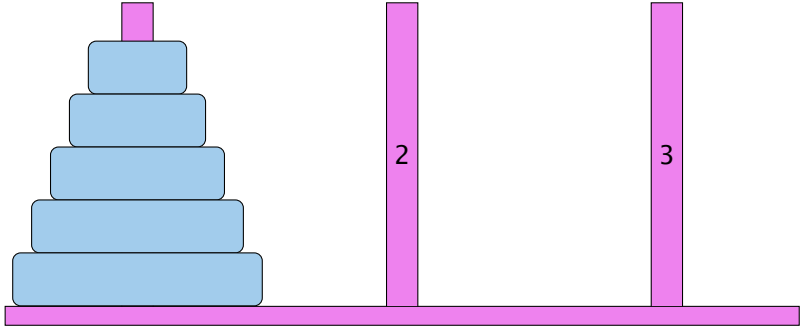
Bemerkung

- Jede Rekursion läßt sich beseitigen, indem man den Aufruf-Keller **explizit** verwaltet.
- Nur im Falle von Endrekursion kann man auf den Keller verzichten.
- Rekursion ist trotzdem nützlich, weil rekursive Programme oft leichter zu verstehen sind als äquivalente Programme ohne Rekursion...

Kontrollfluss

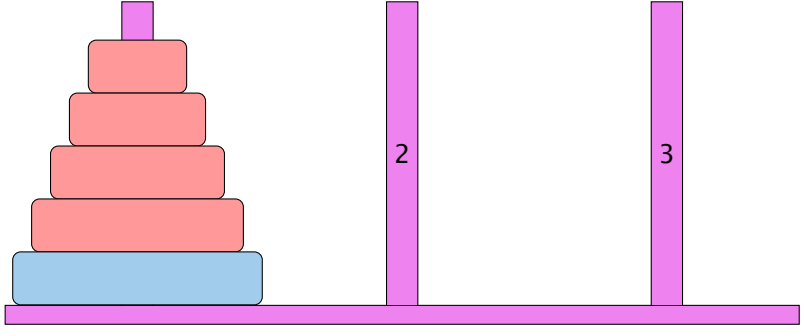


9 Türme von Hanoi



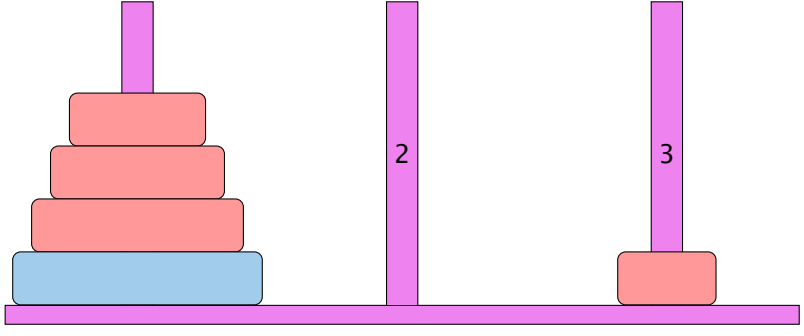
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



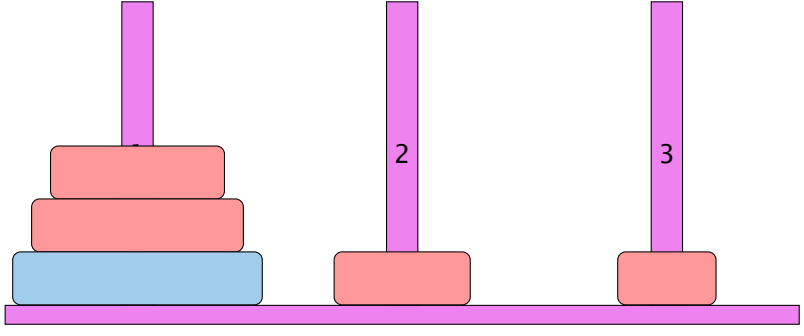
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



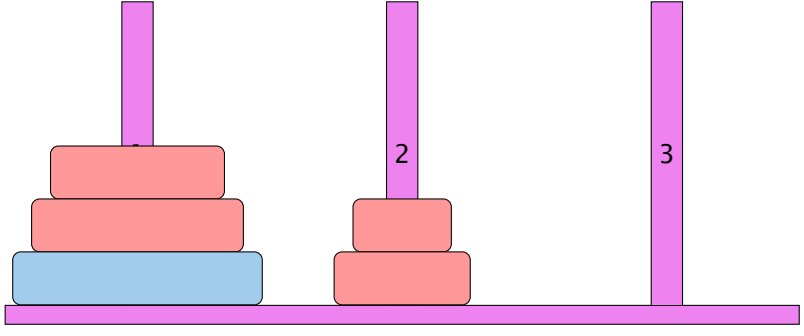
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



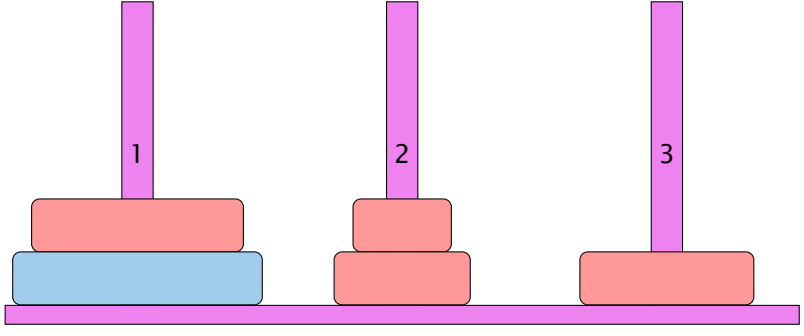
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



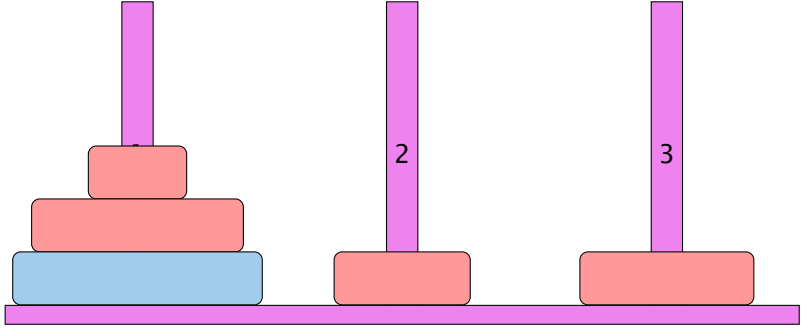
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



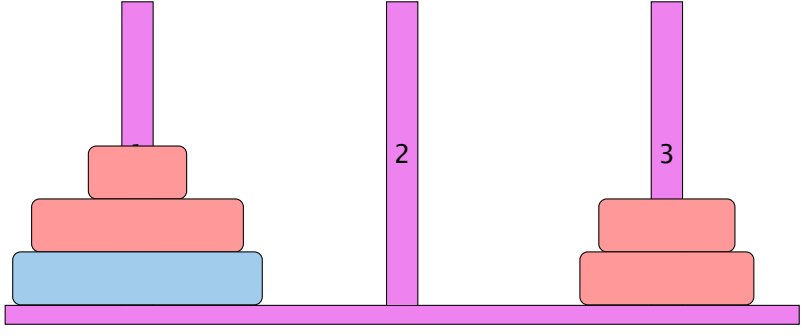
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



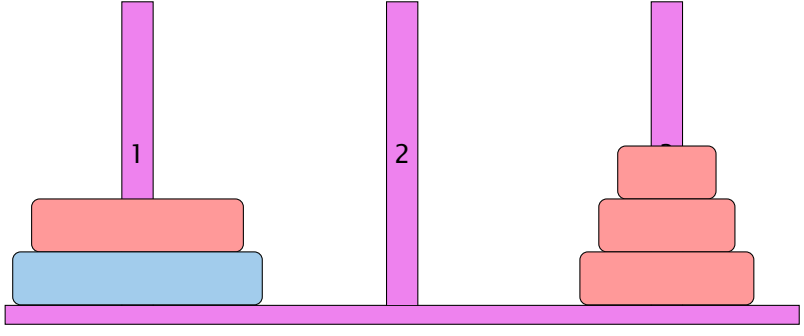
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



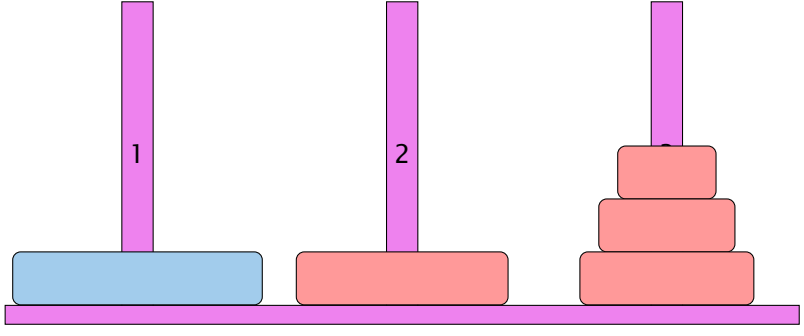
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



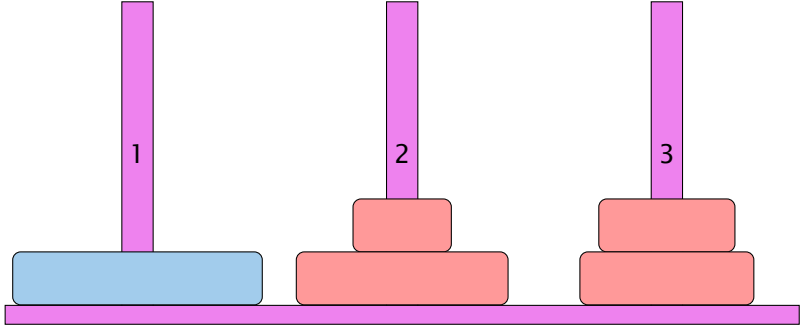
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



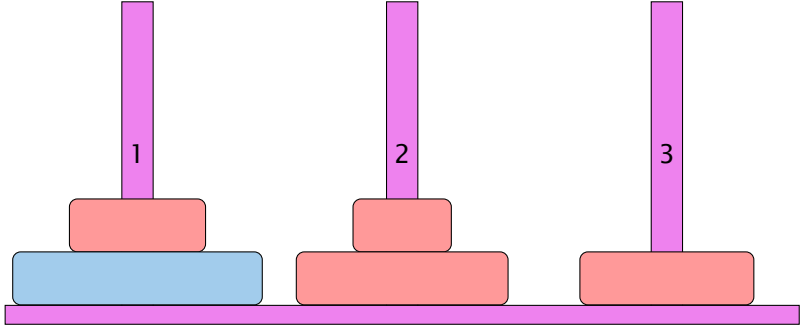
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



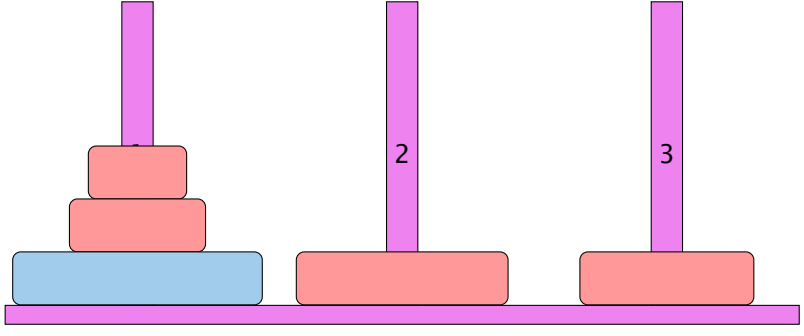
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



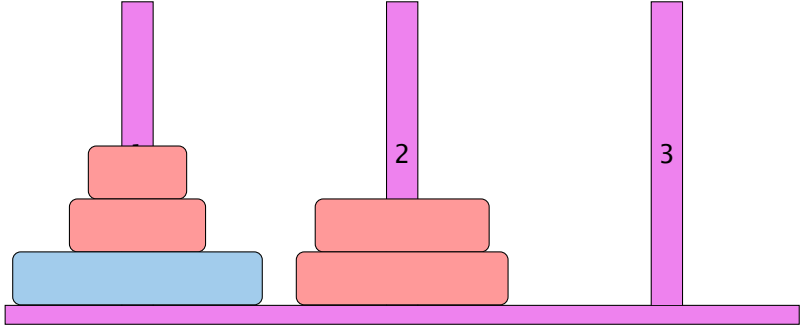
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



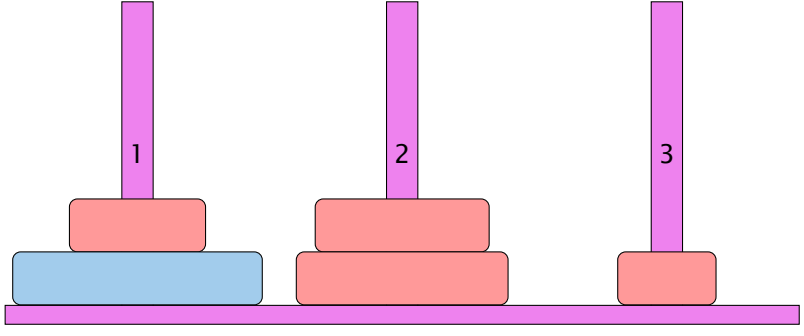
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



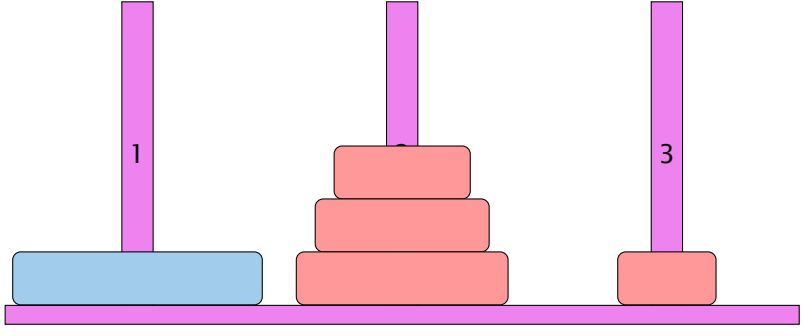
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



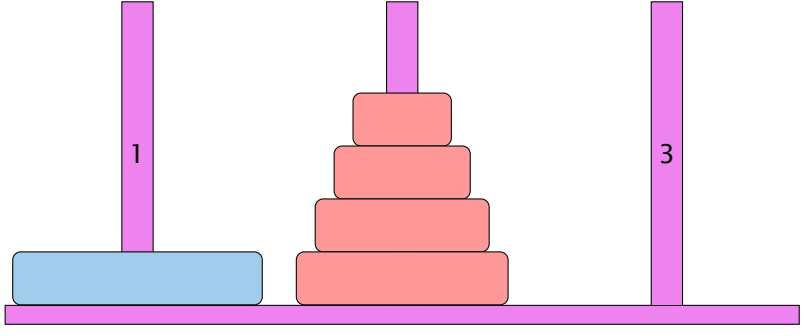
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



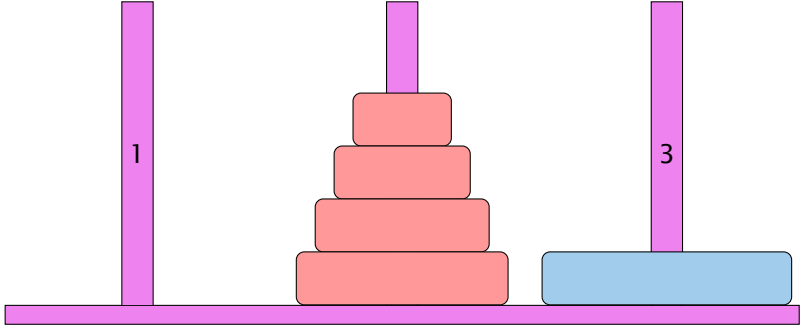
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



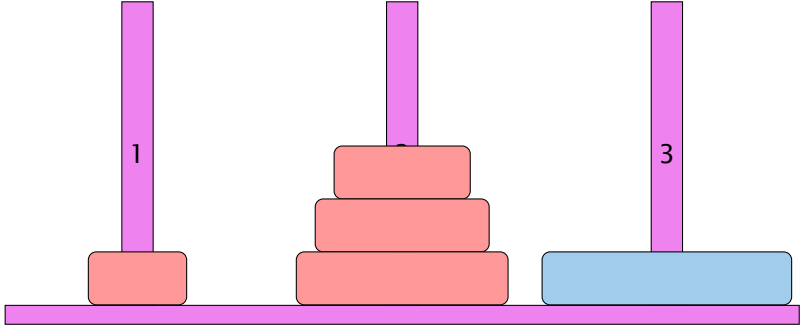
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



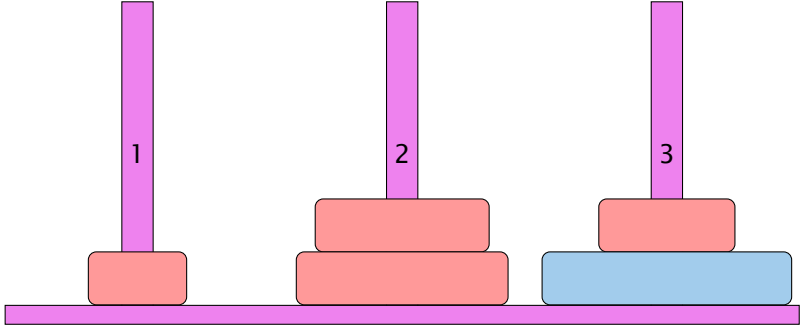
- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi



- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

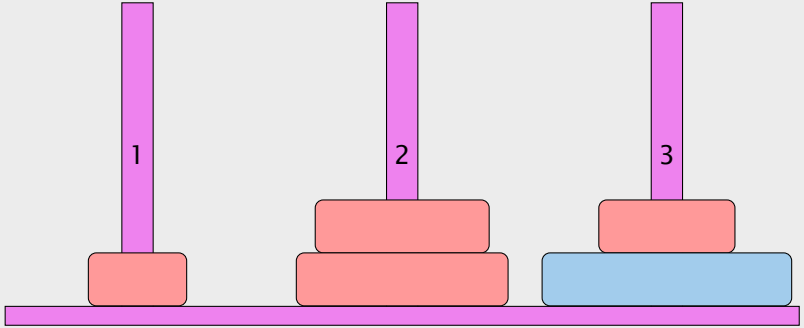
9 Türme von Hanoi

Idee

- Für Turm der Höhe $h = 0$ ist das Problem trivial.
- Falls $h > 0$ zerlegen wir das Problem in drei Teilprobleme:
  1. Versetze oberen $h - 1$ Ringe auf freien Platz
  2. Bewege die unterste Scheibe zum Ziel
  3. Versetze die zwischengelagerten Ringe zum Ziel
- Versetzen eines Turms der Höhe $h > 0$ erfordert also zweimaliges Versetzen eines Turms der Höhe $h - 1$.

Es gibt keine andere Möglichkeit!!!

9 Türme von Hanoi



- Bewege Stapel von links nach rechts.
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer auf einen kleineren Ring gelegt werden.

9 Türme von Hanoi

Idee

- Für Turm der Höhe $h = 0$ ist das Problem trivial.
- Falls $h > 0$ zerlegen wir das Problem in drei Teilprobleme:
  1. Versetze oberen $h - 1$ Ringe auf freien Platz
  2. Bewege die unterste Scheibe zum Ziel
  3. Versetze die zwischengelagerten Ringe zum Ziel
- Versetzen eines Turms der Höhe $h > 0$ erfordert also zweimaliges Versetzen eines Turms der Höhe $h - 1$.

Es gibt keine andere Möglichkeit!!!

Implementierung

```
1 public static void move(int h, byte a, byte b) {
2     if (h > 0) {
3         byte c = free(a,b);
4         move(h-1,a,c);
5         write("move "+a+" to "+b+"\n");
6         move(h-1,c,b);
7     }
8 }
```

. . . bleibt die Ermittlung des freien Rings

Implementierung

```
1 public static void move(int h, byte a, byte b) {
2     if (h > 0) {
3         byte c = free(a,b);
4         move(h-1,a,c);
5         write("move "+a+" to "+b+"\n");
6         move(h-1,c,b);
7     }
8 }
```

. . . bleibt die Ermittlung des freien Rings

Beobachtung

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab. . .

	0	1	2
0		2	1
1	2		0
2	1	0	

free(x,y)

	0	1	2
0		1	2
1	1		3
2	2	3	

sum(x,y)

Beobachtung

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab...

	0	1	2
0		2	1
1	2		0
2	1	0	

free(x,y)

	0	1	2
0		1	2
1	1		3
2	2	3	

sum(x,y)

Implementierung

Um solche Tabellen leicht implementieren zu können stellt Java das `switch`-statement zur Verfügung:

```
1 public static byte free(byte a, byte b) {
2     switch (a + b) {
3         case 1:   return  2;
4         case 2:   return  1;
5         case 3:   return  0;
6         default:  return -1;
7     }
8 }
```

Allgemeines Switch-Statement

```
switch (expr) {
    case const_0: (ss_0)? (break;)?
    case const_1: (ss_1)? (break;)?
                  ...
    case const_{k-1}: (ss_{k-1})? (break;)?
   (default: ss_k)?
}
```

- `expr` sollte eine ganze Zahl/`char`, oder ein `String` sein.
- Die $\texttt{const}_i$ sind Konstanten des gleichen Typs.
- Die $\texttt{ss}_i$ sind alternative Statement-Folgen.
- `default` ist für den Fall, dass keine der Konstanten zutrifft
- Fehlt ein `break`-Statement, wird mit den Statement-Folgen der nächsten Alternative fortgefahren!

Implementierung

Um solche Tabellen leicht implementieren zu können stellt Java das `switch`-statement zur Verfügung:

```
1 public static byte free(byte a, byte b) {
2     switch (a + b) {
3         case 1:   return  2;
4         case 2:   return  1;
5         case 3:   return  0;
6         default:  return -1;
7     }
8 }
```

Beispiel

```
1  int numOfDays;
2  boolean schaltjahr = true;
3  switch (monat) {
4      case "April":
5      case "Juni":
6      case "September":
7      case "November":
8          numOfDays = 30;
9          break;
10     case "Februar":
11         if (schaltjahr)
12             numOfDays = 29;
13         else
14             numOfDays = 28;
15         break;
16     default:
17         numOfDays = 31;
18 }
```

Allgemeines Switch-Statement

```
switch (expr) {
    case const_0: (ss_0)? (break;)?
    case const_1: (ss_1)? (break;)?
                  ...
    case const_{k-1}: (ss_{k-1})? (break;)?
   (default: ss_k)?
}
```

- `expr` sollte eine ganze Zahl/`char`, oder ein `String` sein.
- Die $\texttt{const}_i$ sind Konstanten des gleichen Typs.
- Die $\texttt{ss}_i$ sind alternative Statement-Folgen.
- `default` ist für den Fall, dass keine der Konstanten zutrifft
- Fehlt ein `break`-Statement, wird mit den Statement-Folgen der nächsten Alternative fortgefahren!

Der Bedingungsoperator

Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

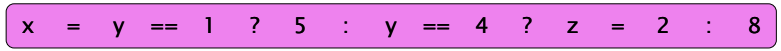
Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
?:	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel

```
1  int numOfDays;
2  boolean schaltjahr = true;
3  switch (monat) {
4      case "April":
5      case "Juni":
6      case "September":
7      case "November":
8          numOfDays = 30;
9          break;
10     case "Februar":
11         if (schaltjahr)
12             numOfDays = 29;
13         else
14             numOfDays = 28;
15         break;
16     default:
17         numOfDays = 31;
18 }
```

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8

Der Bedingungsoperator

Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8

Der Bedingungsoperator

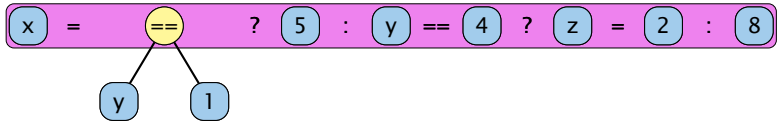
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

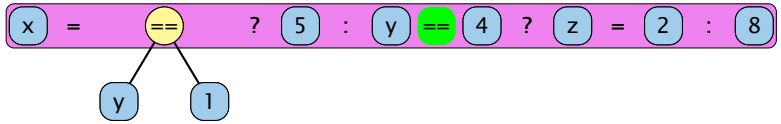
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

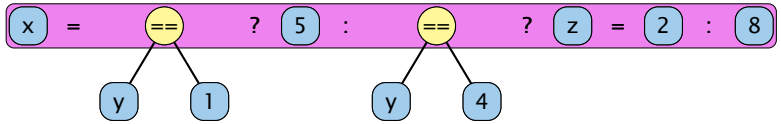
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

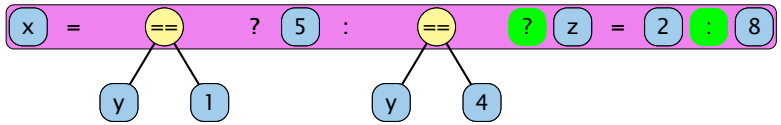
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

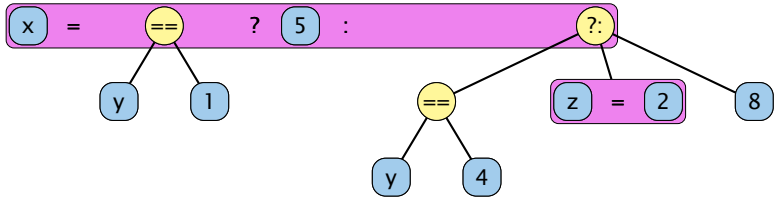
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

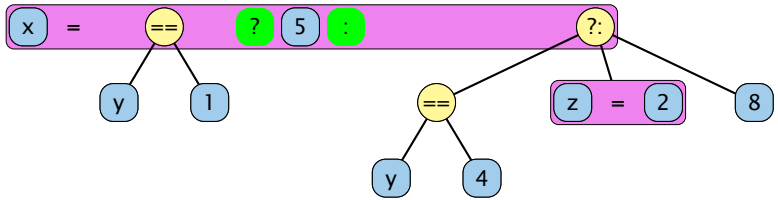
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

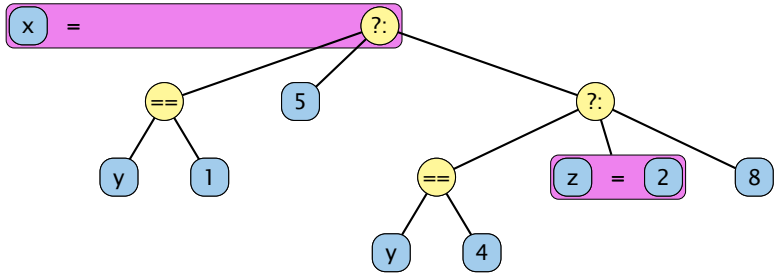
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

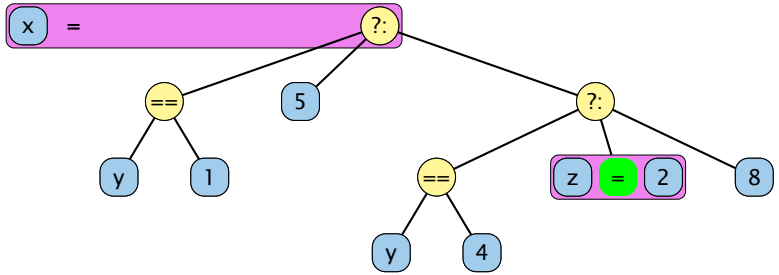
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

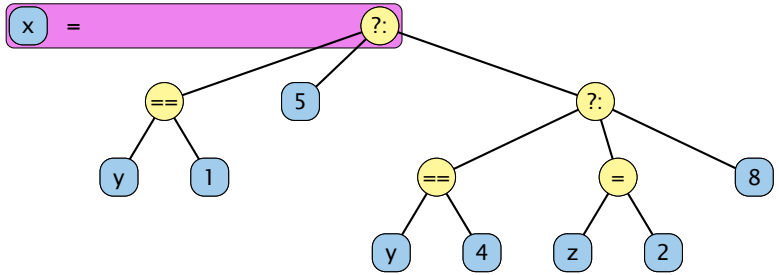
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

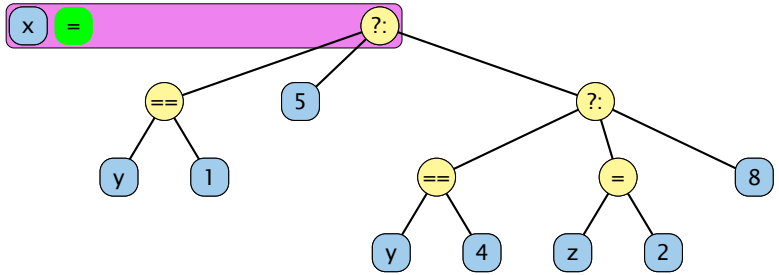
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

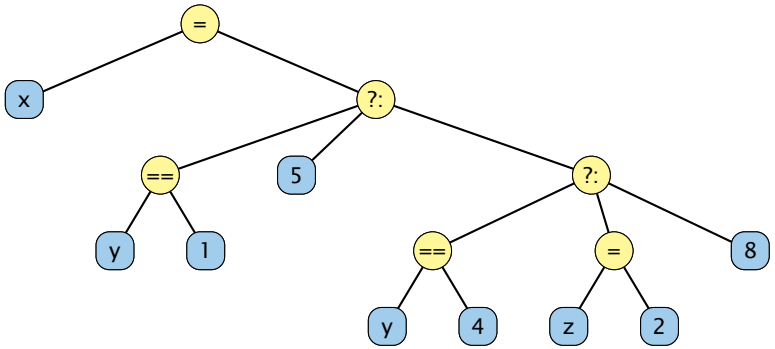
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Der Bedingungsoperator

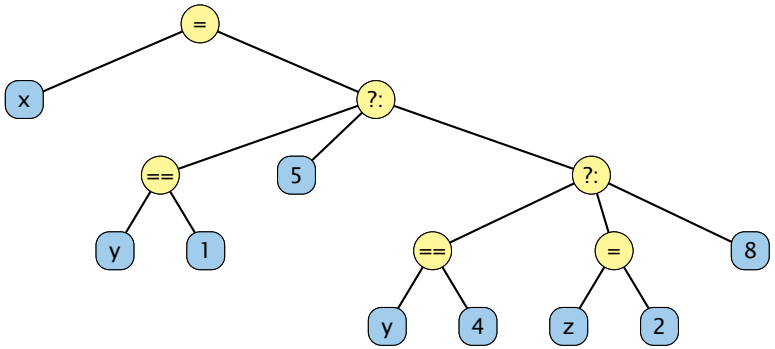
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

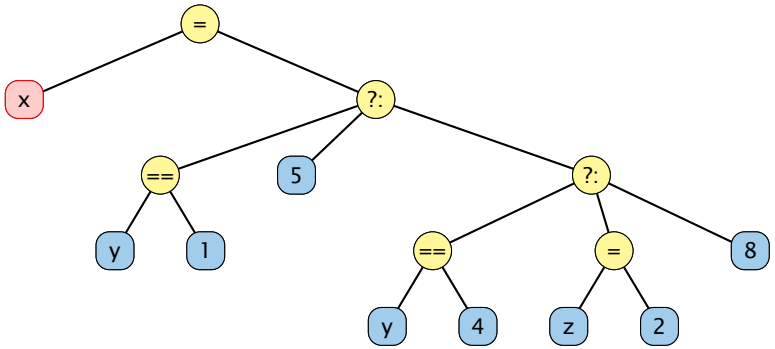
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition `?` expr1 `:` expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
`?:`	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

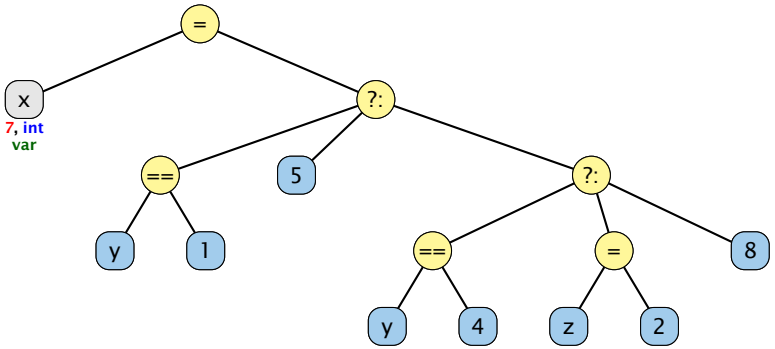
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

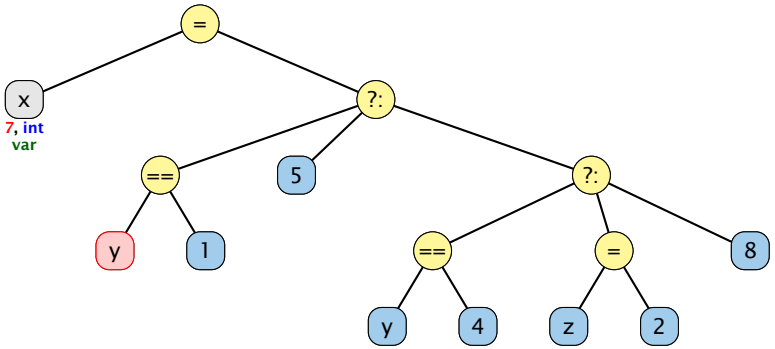
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

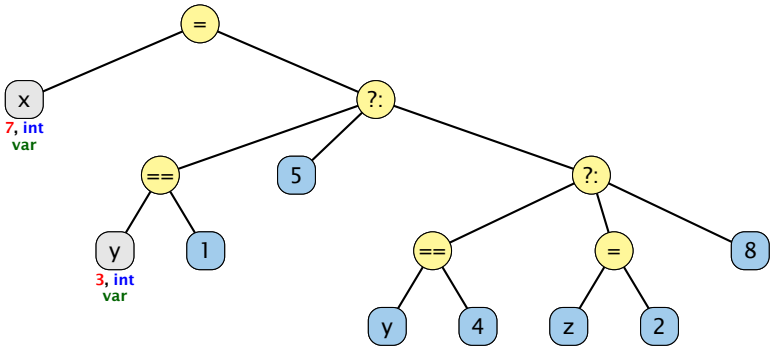
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

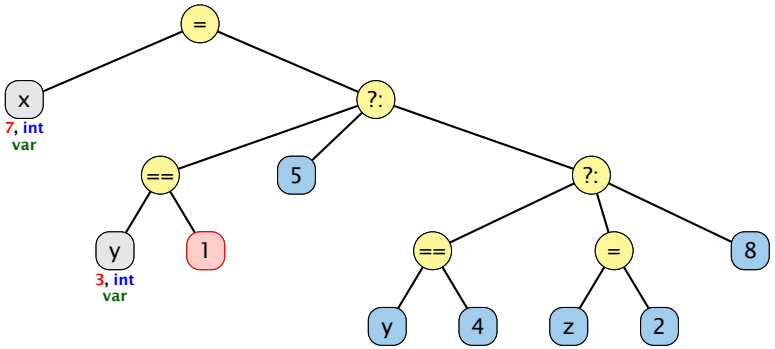
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

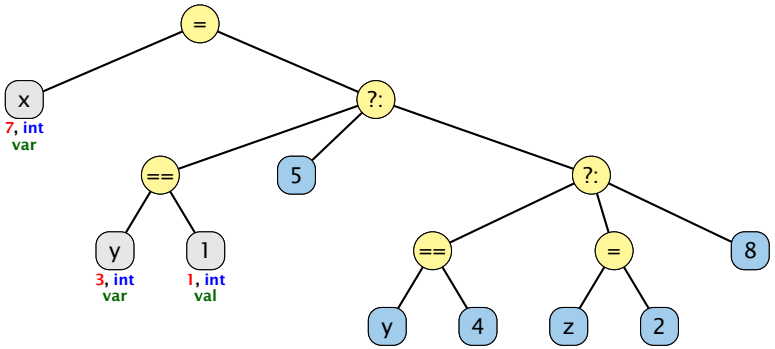
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

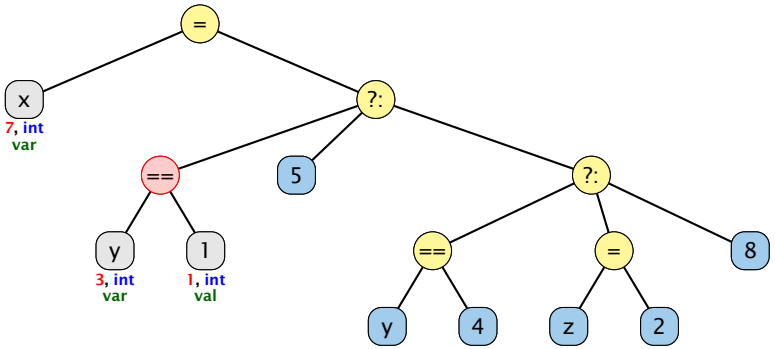
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

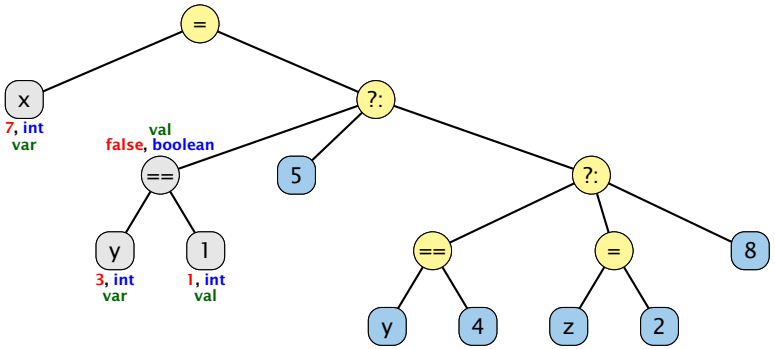
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

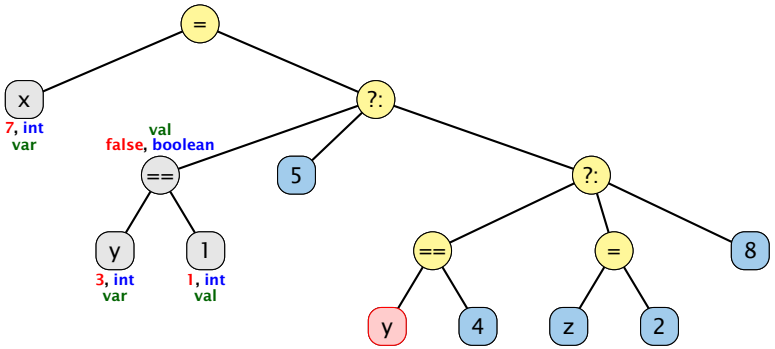
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

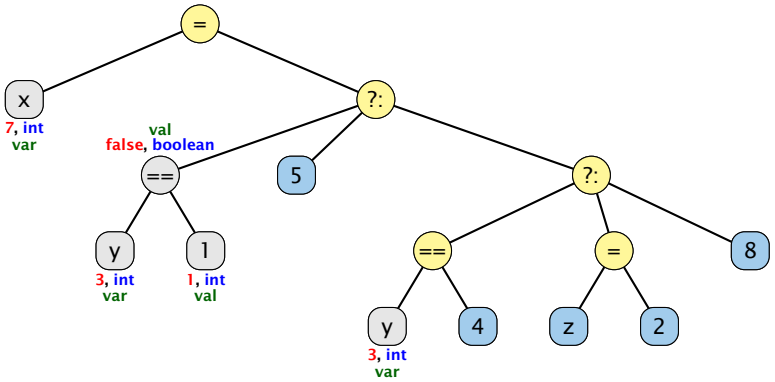
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

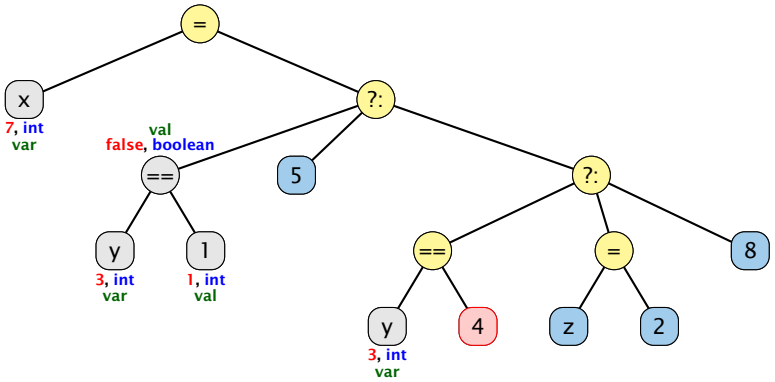
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

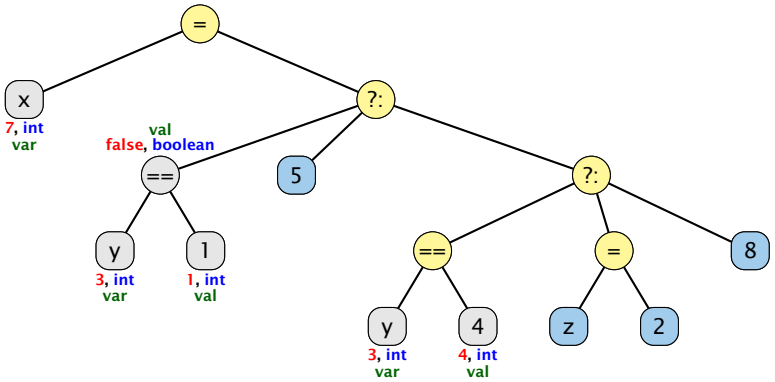
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

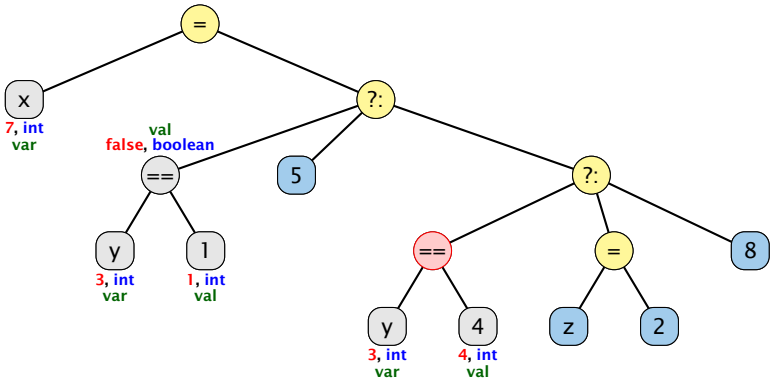
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

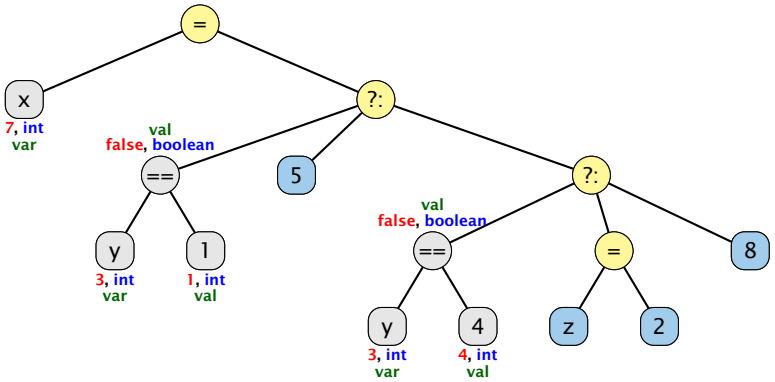
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

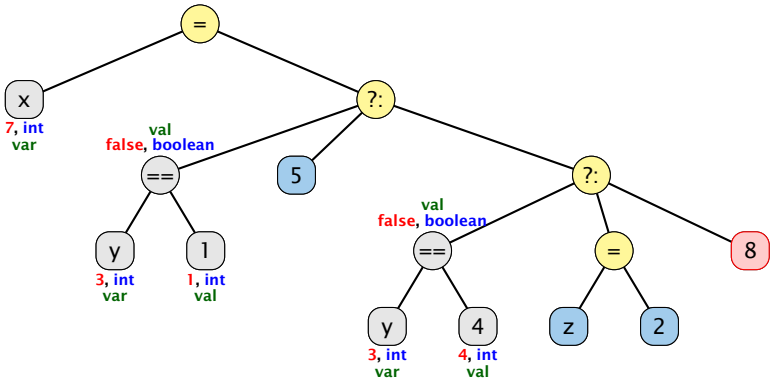
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

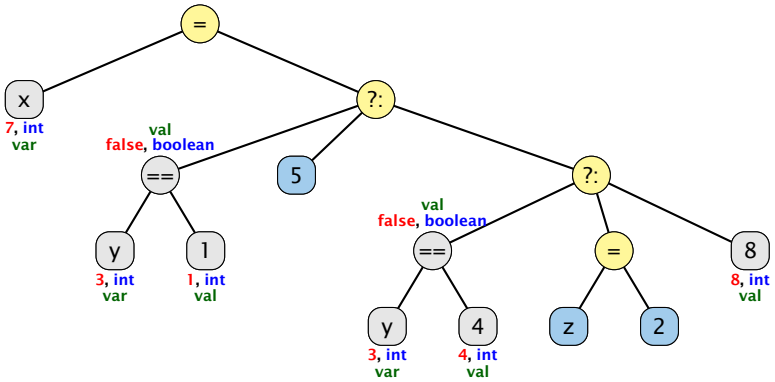
Eine Alternative zu einem switch ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

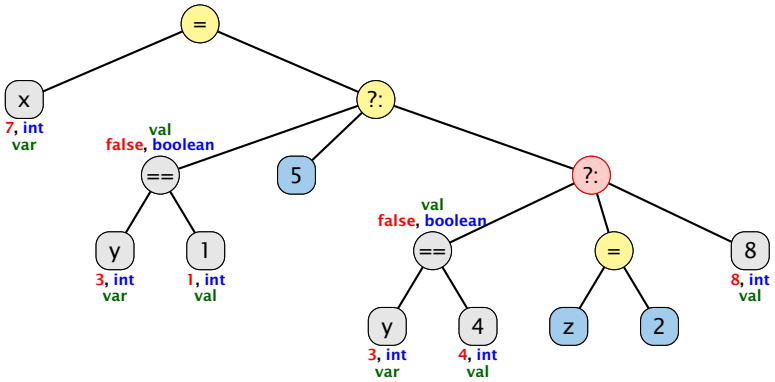
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

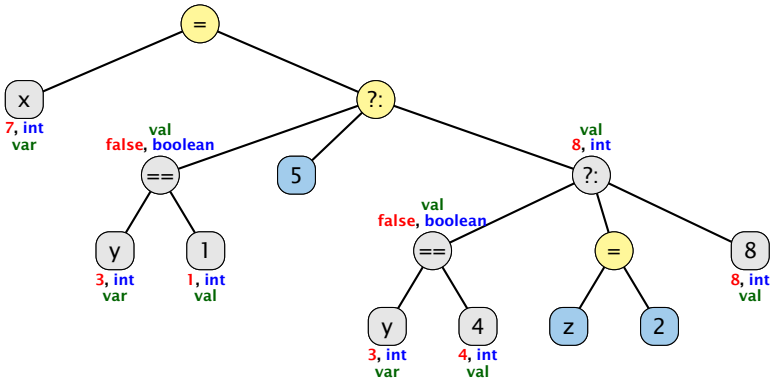
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

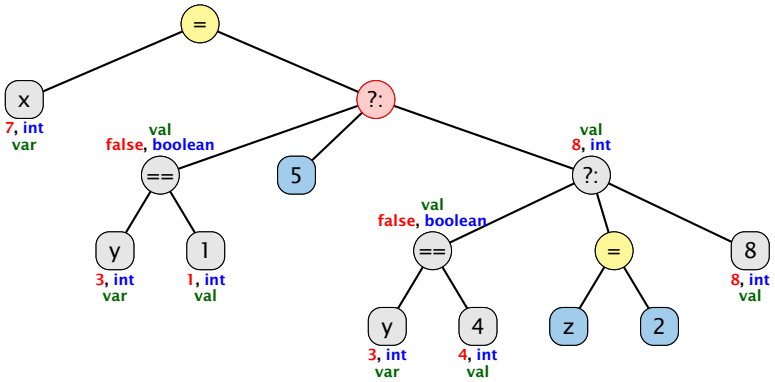
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

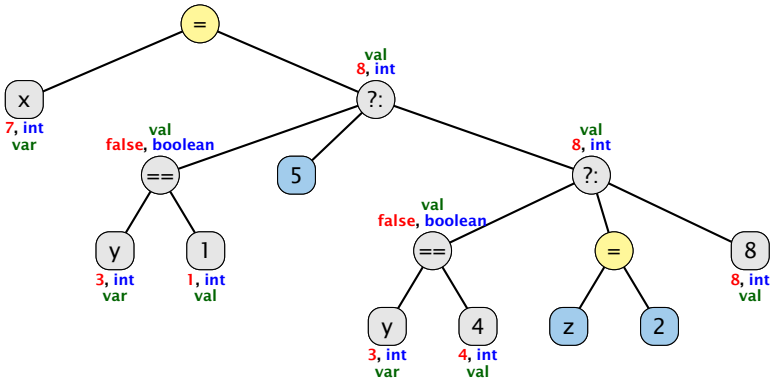
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

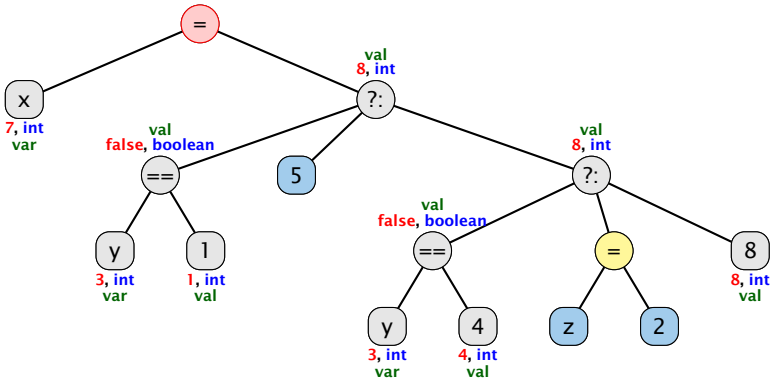
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 7 y 3 z 5

Der Bedingungsoperator

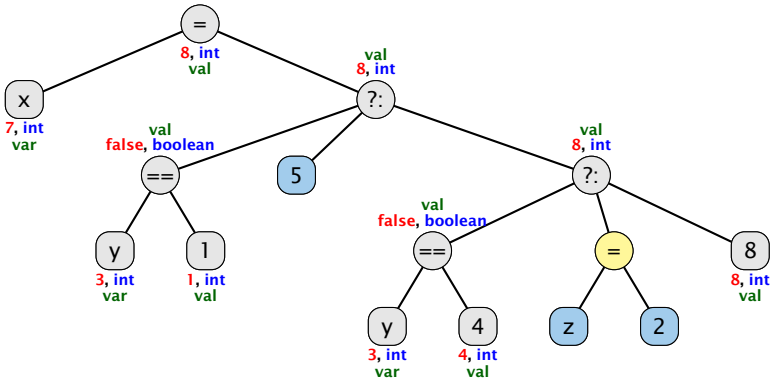
Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



x 8 y 3 z 5

Der Bedingungsoperator

Eine Alternative zu einem `switch` ist der Bedingungsoperator:

condition ? expr1 : expr2

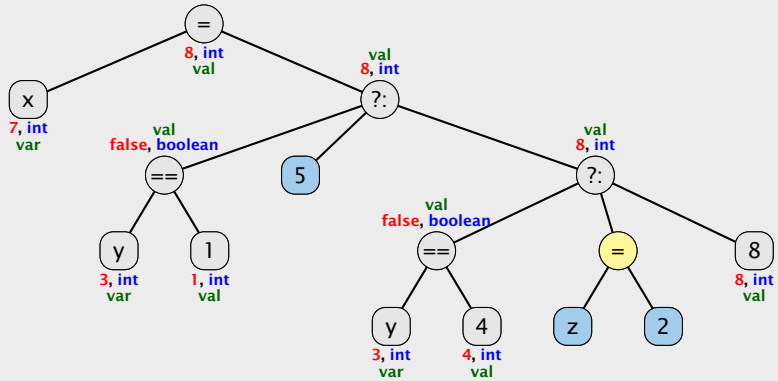
Der Bedingungsoperator

symbol	*name*	*types*	*L/R*	*level*
? :	Bedingungsoperator	boolean, 2*typ	rechts	14

Beispiel

```
numOfDays =
        "Januar".equals(monat) ? 31 :
       "Februar".equals(monat) ? (schaltjahr ? 29 : 28) :
         "Maerz".equals(monat) ? 31 :
         "April".equals(monat) ? 30 :
           "Mai".equals(monat) ? 31 :
          "Juni".equals(monat) ? 30 :
          "Juli".equals(monat) ? 31 :
        "August".equals(monat) ? 31 :
     "September".equals(monat) ? 30 :
       "Oktober".equals(monat) ? 31 :
      "November".equals(monat) ? 30 :
      "Dezember".equals(monat) ? 31 :
                                 -1 ;
```

Beispiel: x = y == 1 ? 5 : y == 4 ? z = 2 : 8



Implementierung

Für unseren Fall geht das viel einfacher:

```
1 public static byte free(byte a, byte b) {
2     return (byte) (3-(a+b));
3 }
```

Beispiel

```
numOfDays =
        "Januar".equals(monat) ? 31 :
      "Februar".equals(monat) ? (schaltjahr ? 29 : 28) :
        "Maerz".equals(monat) ? 31 :
        "April".equals(monat) ? 30 :
          "Mai".equals(monat) ? 31 :
         "Juni".equals(monat) ? 30 :
         "Juli".equals(monat) ? 31 :
       "August".equals(monat) ? 31 :
    "September".equals(monat) ? 30 :
      "Oktober".equals(monat) ? 31 :
     "November".equals(monat) ? 30 :
     "Dezember".equals(monat) ? 31 :
                                -1 ;
```

Bemerkungen:

- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

$$N(h) = \begin{cases} 0 & \text{für } h = 0 \\ 1 + 2N(h-1) & \text{andernfalls} \end{cases}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden.

  Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter...

Implementierung

Für unseren Fall geht das viel einfacher:

```
1 public static byte free(byte a, byte b) {
2     return (byte) (3-(a+b));
3 }
```

Bis jetzt kennen wir (fast) nur primitive Datentypen.

Diese entsprechen weitestgehend der Hardware des Rechners (z.B. besitzt ein Rechner Hardware um zwei `floats` zu addieren).

Wir möchten Dinge der realen Welt modellieren, dafür benötigen wir komplexere Datentypen.

Lösung: selbstdefinierte Datentypen

Objektorientierte Programmierung

Angenommen wir möchten eine Adressverwaltung schreiben.
Dazu müßten wir zunächst eine Adresse modellieren:

Harald Räcke
Boltzmannstraße 3
85748 Garching

Adresse	
+ Name	: String
+ Strasse	: String
+ Hausnummer	: int
+ Postleitzahl	: int
+ Stadt	: String

Zumindest für diesen Fall, ist die Modellierung sehr einfach.

Datentyp ist hier nur eine Komposition (Zusammensetzung) von anderen einfacheren Grundtypen.

Objektorientierte Programmierung

Bis jetzt kennen wir (fast) nur primitive Datentypen.

Diese entsprechen weitestgehend der Hardware des Rechners (z.B. besitzt ein Rechner Hardware um zwei `floats` zu addieren).

Wir möchten Dinge der realen Welt modellieren, dafür benötigen wir komplexere Datentypen.

Lösung: selbstdefinierte Datentypen

Objektorientierte Programmierung

Angenommen wir möchten eine Adressverwaltung schreiben.
Dazu müßten wir zunächst eine Adresse modellieren:

Harald Räcke
Boltzmannstraße 3
85748 Garching

Adresse	
+ Name	: String
+ Strasse	: String
+ Hausnummer	: int
+ Postleitzahl	: int
+ Stadt	: String

Zumindest für diesen Fall, ist die Modellierung sehr einfach.

Datentyp ist hier nur eine Komposition (Zusammensetzung) von anderen einfacheren Grundtypen.

Objektorientierte Programmierung

Bis jetzt kennen wir (fast) nur primitive Datentypen.

Diese entsprechen weitestgehend der Hardware des Rechners (z.B. besitzt ein Rechner Hardware um zwei `floats` zu addieren).

Wir möchten Dinge der realen Welt modellieren, dafür benötigen wir komplexere Datentypen.

Lösung: selbstdefinierte Datentypen

Objektorientierte Programmierung

Wie benutzt man den Datentyp?

Geht aus der Ansammlung der Grundtypen nicht hervor. Wenn der Datentyp sehr komplex ist (Atomreaktor), kann man leicht Fehler machen, und einen ungültigen Zustand erzeugen.

Objektorientierte Programmierung

Angenommen wir möchten eine Adressverwaltung schreiben. Dazu müßten wir zunächst eine Adresse modellieren:

Harald Räcke
Boltzmannstraße 3
85748 Garching

Adresse	
+ Name	: String
+ Strasse	: String
+ Hausnummer	: int
+ Postleitzahl	: int
+ Stadt	: String

Zumindest für diesen Fall, ist die Modellierung sehr einfach.

Datentyp ist hier nur eine Komposition (Zusammensetzung) von anderen einfacheren Grundtypen.

Objektorientierte Programmierung

Wie benutzt man den Datentyp?

Geht aus der Ansammlung der Grundtypen nicht hervor. Wenn der Datentyp sehr komplex ist (Atomreaktor), kann man leicht Fehler machen, und einen ungültigen Zustand erzeugen.

Grundidee:
Ändere Variablen des Datentyps nur über Funktionen/Methoden.

Falls diese korrekt implementiert sind, kann man keinen ungültigen Zustand erzeugen.

Daten und Methoden
gehören zusammen
(abstrakter Datentyp)

Objektorientierte Programmierung

Angenommen wir möchten eine Adressverwaltung schreiben.
Dazu müßten wir zunächst eine Adresse modellieren:

Harald Räcke
Boltzmannstraße 3
85748 Garching

Adresse	
+ Name	: String
+ Strasse	: String
+ Hausnummer	: int
+ Postleitzahl	: int
+ Stadt	: String

Zumindest für diesen Fall, ist die Modellierung sehr einfach.

Datentyp ist hier nur eine Komposition (Zusammensetzung) von anderen einfacheren Grundtypen.

Objektorientierte Programmierung

Ein (abstrakter) Datentyp besteht aus Daten und einer Menge von Methoden (Schnittstelle) um diese Daten zu manipulieren.

Datenkapselung / Information Hiding

Die Implementierung des Datentyps wird vor dem Benutzer versteckt.

- minimiert Fehler durch unsachgemäßen Zugriff
- Entkopplung von Teilproblemen
  - gut für Implementierung, aber auch
  - Fehlersuche und Wartung
- erlaubt es die Implementierung später anzupassen (↑rapid prototyping)
- erzwingt in der Designphase über das was und nicht über das wie nachzudenken....

Objektorientierte Programmierung

Wie benutzt man den Datentyp?

Geht aus der Ansammlung der Grundtypen nicht hervor. Wenn der Datentyp sehr komplex ist (Atomreaktor), kann man leicht Fehler machen, und einen ungültigen Zustand erzeugen.

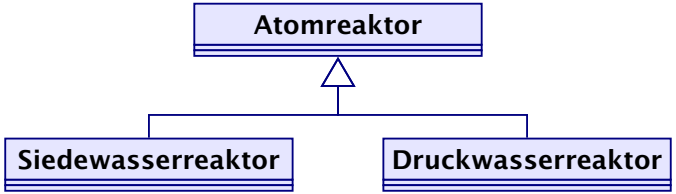
Grundidee:
Ändere Variablen des Datentyps nur über Funktionen/Methoden.

Falls diese korrekt implementiert sind, kann man keinen ungültigen Zustand erzeugen.

Daten und Methoden
gehören zusammen
(abstrakter Datentyp)

Generalisierung + Vererbung

Identifiziere Ähnlichkeiten zwischen Datentypen und lagere gemeinsame Teile in einen anderen Datentyp aus.



- vermeidet Copy&Paste...
- verringert den Wartungsaufwand...

Objektorientierte Programmierung

Ein (abstrakter) Datentyp besteht aus Daten und einer Menge von Methoden (Schnittstelle) um diese Daten zu manipulieren.

Datenkapselung / Information Hiding

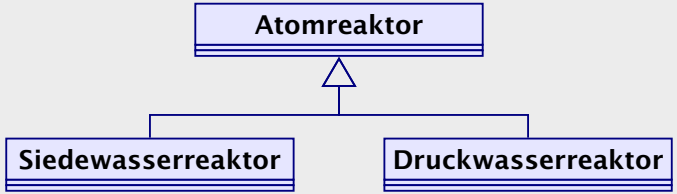
Die Implementierung des Datentyps wird vor dem Benutzer versteckt.

- minimiert Fehler durch unsachgemäßen Zugriff
- Entkopplung von Teilproblemen
  - gut für Implementierung, aber auch
  - Fehlersuche und Wartung
- erlaubt es die Implementierung später anzupassen (↑rapid prototyping)
- erzwingt in der Designphase über das was und nicht über das wie nachzudenken....

Klasse = Implementierung eines abstrakten Datentyps

Objekt = Instanz/Variable einer Klasse

Generalisierung + Vererbung
Identifiziere Ähnlichkeiten zwischen Datentypen und lagere gemeinsame Teile in einen anderen Datentyp aus.



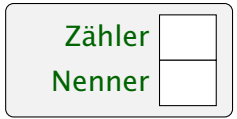
- vermeidet Copy&Paste...
- verringert den Wartungsaufwand...

Objektorientierte Programmierung

Klasse	=	Implementierung eines abstrakten Datentyps
Objekt	=	Instanz/Variable einer Klasse

Beispiel: Rationale Zahlen

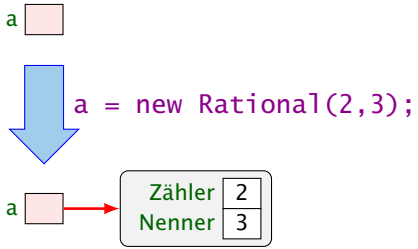
- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $\frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q.
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` erhalten:



- Die Daten eines Objektes heißen Instanz-Variablen oder Attribute.

Beispiel: Rationale Zahlen

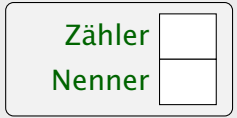
- `Rational name;` deklariert eine Variable für Objekte der Klasse `Rational`.
- Das Kommando `new Rational(...)` legt das Objekt an, ruft einen Konstruktor für dieses Objekt auf, und liefert einen Verweis auf das neue Objekt zurück.



- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objektes initialisieren kann.

Beispiel: Rationale Zahlen

- Eine rationale Zahl $q \in \mathbb{Q}$ hat die Form $\frac{x}{y}$, wobei $x, y \in \mathbb{Z}$.
- x und y heißen Zähler und Nenner von q.
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` erhalten:

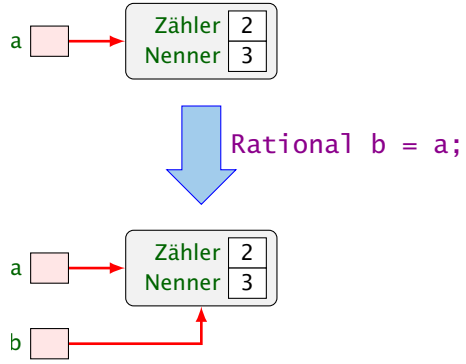


- Die Daten eines Objektes heißen Instanz-Variablen oder Attribute.

Referenzen

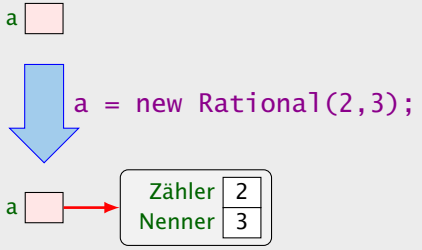
Der Wert der `Rational`-Variablen ist eine Referenz/Verweis auf einen Speicherbereich.

`Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:



Beispiel: Rationale Zahlen

- `Rational name;` deklariert eine Variable für Objekte der Klasse `Rational`.
- Das Kommando `new Rational(...)` legt das Objekt an, ruft einen Konstruktor für dieses Objekt auf, und liefert einen Verweis auf das neue Objekt zurück.

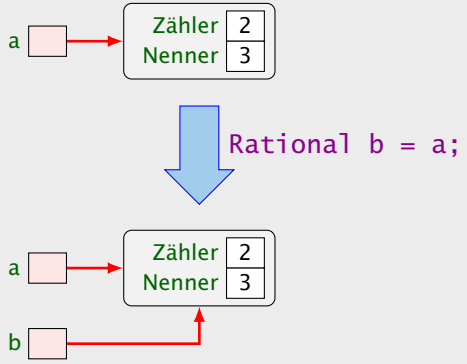


- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objektes initialisieren kann.

Referenzen

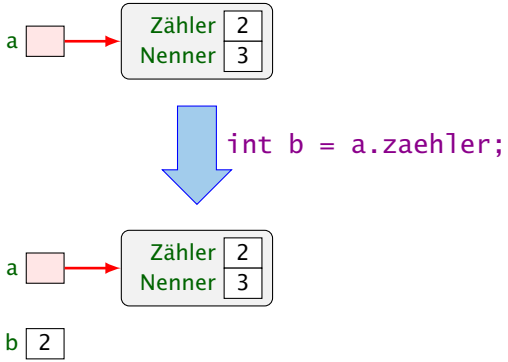
Der Wert der `Rational`-Variablen ist eine Referenz/Verweis auf einen Speicherbereich.

`Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:



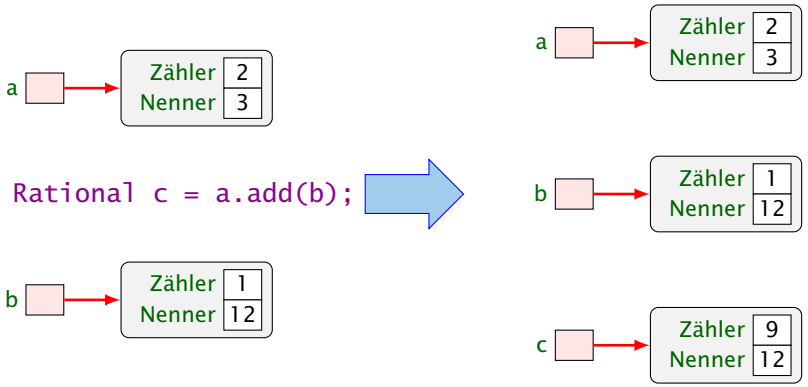
Beispiel: Rationale Zahlen

`a.zaehler` liefert den Wert des Attributs `zaehler` des Objektes auf das `a` verweist:



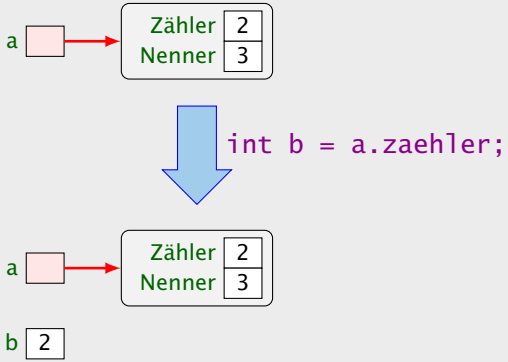
Beispiel: Rationale Zahlen

`a.add(b)` ruft die Operation `add` für `a` mit dem zusätzlichen aktuellen Parameter `b` auf:

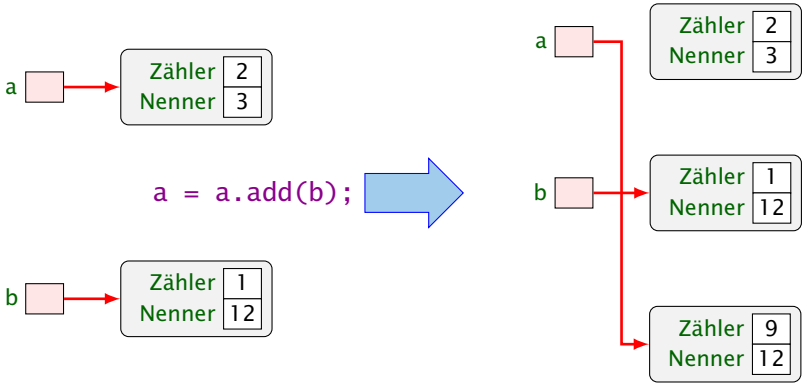


Beispiel: Rationale Zahlen

`a.zaehler` liefert den Wert des Attributs `zaehler` des Objektes auf das `a` verweist:



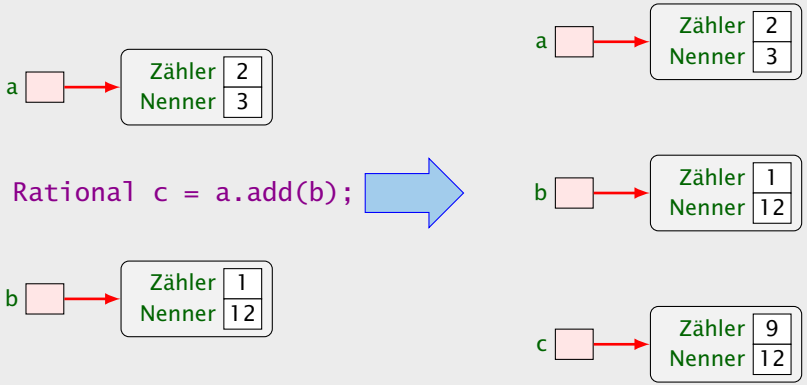
Beispiel: Rationale Zahlen



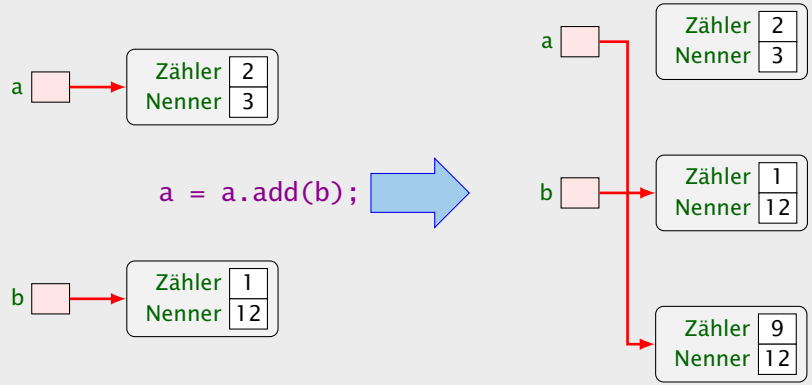
Die Operationen auf Objekten einer Klasse heißen auch Methoden, genauer: Objekt-Methoden.

Beispiel: Rationale Zahlen

`a.add(b)` ruft die Operation `add` für `a` mit dem zusätzlichen aktuellen Parameter `b` auf:



Beispiel: Rationale Zahlen



Die Operationen auf Objekten einer Klasse heißen auch Methoden, genauer: Objekt-Methoden.

Zusammenfassung

Eine Klassendeklaration besteht folglich aus:

- Attributen für die verschiedenen Wertkombinationen der Objekte;
- Konstruktoren zur Initialisierung der Objekte;
- Methoden, d.h. Operationen auf Objekten.

Diese Elemente heißen auch Members der Klasse.

Zusammenfassung

Eine Klassendeklaration besteht folglich aus:

- Attributen für die verschiedenen Wertkombinationen der Objekte;
- Konstruktoren zur Initialisierung der Objekte;
- Methoden, d.h. Operationen auf Objekten.

Diese Elemente heißen auch Members der Klasse.

Implementierung

```
1  public class Rational {
2      // Attribute:
3      private int zaehler, nenner;
4      // Konstruktoren:
5      public Rational(int x, int y) {
6          zaehler = x;
7          nenner  = y;
8      }
9      public Rational(int x) {
10         zaehler = x;
11         nenner  = 1;
12     }
```

Implementierung

```
13      // Objekt-Methoden:
14      public Rational add(Rational r) {
15          int x = zaehler * r.nenner + r.zaehler * nenner;
16          int y = nenner * r.nenner;
17          return new Rational(x,y);
18      }
19      public boolean isEqual(Rational r) {
20          return zaehler * r.nenner == r.zaehler * nenner;
21      }
22      public String toString() {
23          if (nenner == 1) return "" + zaehler;
24          if (nenner  > 0) return zaehler +"/"+ nenner;
25          return (-zaehler) +"/"+ (-nenner);
26      }
27      public static Rational[] intToRationalArray(int[] a) {
28          Rational[] b = new Rational[a.length];
29          for(int i=0; i < a.length; ++i)
30              b[i] = new Rational(a[i]);
31          return b;
32      }
```

Implementierung

```
1  public class Rational {
2      // Attribute:
3      private int zaehler, nenner;
4      // Konstruktoren:
5      public Rational(int x, int y) {
6          zaehler = x;
7          nenner  = y;
8      }
9      public Rational(int x) {
10         zaehler = x;
11         nenner  = 1;
12     }
```

Implementierung

```
33      // Jetzt kommt das Hauptprogramm
34      public static void main(String[] args) {
35          Rational a = new Rational(1,2);
36          Rational b = new Rational(3,4);
37          Rational c = a.add(b);
38
39          System.out.println(c.toString());
40      } // end of main()
41  } // end of class Rational
```

Implementierung

```
13      // Objekt-Methoden:
14      public Rational add(Rational r) {
15          int x = zaehler * r.nenner + r.zaehler * nenner;
16          int y = nenner * r.nenner;
17          return new Rational(x,y);
18      }
19      public boolean isEqual(Rational r) {
20          return zaehler * r.nenner == r.zaehler * nenner;
21      }
22      public String toString() {
23          if (nenner == 1) return "" + zaehler;
24          if (nenner  > 0) return zaehler +"/"+ nenner;
25          return (-zaehler) +"/"+ (-nenner);
26      }
27      public static Rational[] intToRationalArray(int[] a) {
28          Rational[] b = new Rational[a.length];
29          for(int i=0; i < a.length; ++i)
30              b[i] = new Rational(a[i]);
31          return b;
32      }
```

Implementierung

Bemerkungen:

- Jede Klasse sollte in einer separaten Datei des entsprechenden Namens stehen.
- Die Schlüsselworte `public` bzw. `private` klassifizieren, für wen die enstprechenden Members sichtbar, d.h. zugänglich sind.
- `private` heißt: nur für Members der gleichen Klasse sichtbar.
- `public` heißt: innerhalb des gesamten Programms sichtbar.
- Nicht klassifizierte Members sind nur innerhalb des aktuellen ↑Package sichtbar.

Implementierung

```
33      // Jetzt kommt das Hauptprogramm
34      public static void main(String[] args) {
35          Rational a = new Rational(1,2);
36          Rational b = new Rational(3,4);
37          Rational c = a.add(b);
38
39          System.out.println(c.toString());
40      } // end of main()
41  } // end of class Rational
```

Implementierung

Bemerkungen:

- Jede Klasse sollte in einer separaten Datei des entsprechenden Namens stehen.
- Die Schlüsselworte `public` bzw. `private` klassifizieren, für wen die enstprechenden Members sichtbar, d.h. zugänglich sind.
- `private` heißt: nur für Members der gleichen Klasse sichtbar.
- `public` heißt: innerhalb des gesamten Programms sichtbar.
- Nicht klassifizierte Members sind nur innerhalb des aktuellen ↑Package sichtbar.

Excursion – Innere Klassen

In Java kann man auch Klassen innerhalb einer anderen Klasse definieren:

```
1 public class A {
2     private int var;
3     private class B {
4         B() {
5             var = 5;
6         }
7     }
8 }
```

Innerhalb der Klasse B kann man auf alle Members der Klasse A zugreifen.

In diesem Fall kann die Klasse B nicht von aussen zugegriffen werden, da sie `private` ist.

Excursion – Innere Klassen

In Java kann man auch Klassen innerhalb einer anderen Klasse definieren:

```
1 public class A {
2     private int var;
3     private class B {
4         B() {
5             var = 5;
6         }
7     }
8 }
```

Innerhalb der Klasse B kann man auf alle Members der Klasse A zugreifen.

In diesem Fall kann die Klasse B nicht von aussen zugegriffen werden, da sie private ist.

Implementierung

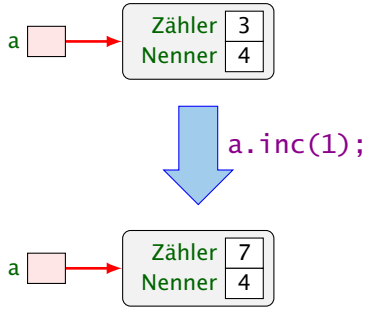
Bemerkungen:

- Konstruktoren haben den gleichen Namen wie die Klasse.
- Es kann mehrere geben, sofern sie sich im Typ ihrer Argumente unterscheiden.
- Konstruktoren haben keine Rückgabewerte und darum auch keinen Rückgabetyp.
- Methoden haben dagegen stets einen Rückgabe-Typ, evt. void.

```
1 public void inc(int b) {
2     zaehler = zaehler + b * nenner;
3 }
```

Implementierung

Die Objekt-Methode `inc()` modifiziert das Objekt, für das sie aufgerufen wird.



Implementierung

Bemerkungen:

- Konstruktoren haben den gleichen Namen wie die Klasse.
- Es kann mehrere geben, sofern sie sich im Typ ihrer Argumente unterscheiden.
- Konstruktoren haben keine Rückgabewerte und darum auch keinen Rückgabetyp.
- Methoden haben dagegen stets einen Rückgabe-Typ, evt. `void`.

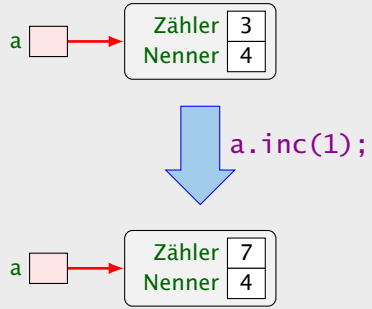
```
1 public void inc(int b) {
2     zaehler = zaehler + b * nenner;
3 }
```

Implementierung

- Die Objektmethode `isEqual()` ist nötig, da der Operator `==` bei Objekten die Identität der Objekte testet, d.h. die Gleichheit der Referenz!!!
- Die Objektmethode `toString()` liefert eine `String`-Darstellung des Objekts.
- Sie wird implizit aufgerufen, wenn das Objekt als Argument für die Konkatenation `+` auftaucht.
- Innerhalb einer Objektmethode/eines Konstruktors kann auf die Attribute des Objektes direkt zugegriffen werden.
- `private`-Klassifizierung bezieht sich auf die Klasse nicht das Objekt: die Attribute aller `Rational`-Objekte sind für `add` sichtbar!!!

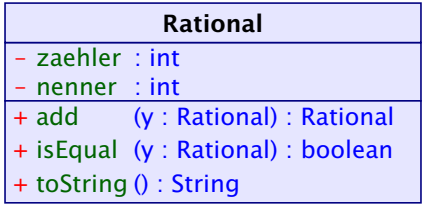
Implementierung

Die Objekt-Methode `inc()` modifiziert das Objekt, für das sie aufgerufen wird.



UML-Diagramm

Eine graphische Visualiserung der Klasse `Rational`, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:

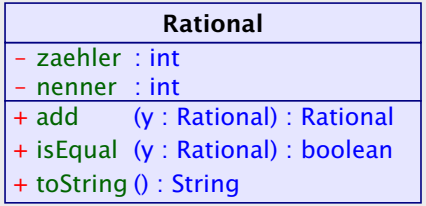


Implementierung

- Die Objektmethode `isEqual()` ist nötig, da der Operator `==` bei Objekten die Identität der Objekte testet, d.h. die Gleichheit der Referenz!!!
- Die Objektmethode `toString()` liefert eine `String`-Darstellung des Objekts.
- Sie wird implizit aufgerufen, wenn das Objekt als Argument für die Konkatenation `+` auftaucht.
- Innerhalb einer Objektmethode/eines Konstruktors kann auf die Attribute des Objektes direkt zugegriffen werden.
- `private`-Klassifizierung bezieht sich auf die Klasse nicht das Objekt: die Attribute aller `Rational`-Objekte sind für `add` sichtbar!!!

UML-Diagramm

Eine graphische Visualiserung der Klasse `Rational`, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:



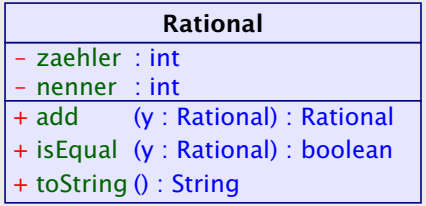
Diskussion und Ausblick

- Solche Diagramme werden von der UML, d.h. der Unified Modelling Language, bereitgestellt, um Software-Systeme zu entwerfen (↑Software Engineering)
- Für einzelne Klassen lohnt sich ein solches Diagramm nicht wirklich.
- Besteht ein System aber aus sehr vielen Klassen, kann man damit die Beziehungen zwischen den Klassen verdeutlichen.

Achtung:
UML wurde nicht speziell für Java entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht oder nur schlecht modellieren.

UML-Diagramm

Eine graphische Visualiserung der Klasse `Rational`, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:



Diskussion und Ausblick

- Solche Diagramme werden von der UML, d.h. der Unified Modelling Language, bereitgestellt, um Software-Systeme zu entwerfen (↑Software Engineering)
- Für einzelne Klassen lohnt sich ein solches Diagramm nicht wirklich.
- Besteht ein System aber aus sehr vielen Klassen, kann man damit die Beziehungen zwischen den Klassen verdeutlichen.

Achtung:
UML wurde nicht speziell für Java entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht oder nur schlecht modellieren.

Diskussion und Ausblick

- Solche Diagramme werden von der UML, d.h. der Unified Modelling Language, bereitgestellt, um Software-Systeme zu entwerfen (↑Software Engineering)
- Für einzelne Klassen lohnt sich ein solches Diagramm nicht wirklich.
- Besteht ein System aber aus sehr vielen Klassen, kann man damit die Beziehungen zwischen den Klassen verdeutlichen.

Achtung:
UML wurde nicht speziell für Java entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht oder nur schlecht modellieren.

10.1 Selbstreferenzen

```
1  public class Cyclic {
2      private int info;
3      private Cyclic ref;
4      // Konstruktor
5      public Cyclic() {
6          info = 17;
7          ref = this;
8      }
9           ...
10 } // end of class Cyclic
```

Innerhalb eines Members kann man mit Hilfe von `this` auf das aktuelle Objekt selbst zugreifen!

10.1 Selbstreferenzen

```
1  public class Cyclic {
2      private int info;
3      private Cyclic ref;
4      // Konstruktor
5      public Cyclic() {
6          info = 17;
7          ref = this;
8      }
9           ...
10 } // end of class Cyclic
```

Innerhalb eines Members kann man mit Hilfe von `this` auf das aktuelle Objekt selbst zugreifen!

10.1 Selbstreferenzen

Für `Cyclic a = new Cyclic();` ergibt das

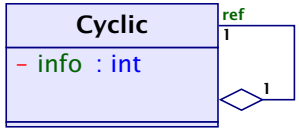
a 

10.1 Selbstreferenzen

Für `Cyclic a = new Cyclic();` ergibt das

a 

Modellierung einer Selbstreferenz

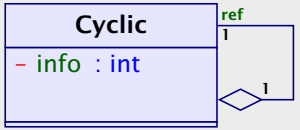


Die Rautenverbindung heißt auch Aggregation

Das Klassendiagramm vermerkt, dass jedes Objekt der Klasse `Cyclic` einen Verweis mit dem Namen ref auf ein weiteres Objekt der Klasse `Cyclic` enhält.

Ausserdem, dass jedes `Cyclic`-Objekt in genau einem anderen `Cyclic`-Objekt, die Rolle ref übernimmt.

Modellierung einer Selbstreferenz



Die Rautenverbindung heißt auch Aggregation

Das Klassendiagramm vermerkt, dass jedes Objekt der Klasse `Cyclic` einen Verweis mit dem Namen ref auf ein weiteres Objekt der Klasse `Cyclic` enhält.

Ausserdem, dass jedes `Cyclic`-Objekt in genau einem anderen `Cyclic`-Objekt, die Rolle ref übernimmt.

Die this-Referenz

Woher kommt die Referenz `this`?

- Einem Auruf einer Objektmethode (z.B. `a.inc()`) oder eines Konstruktors wird implizit ein versteckter Parameter übergeben, der auf das Objekt (hier `a`) zeigt.
- Die Signatur von `inc(int x)` ist eigentlich:

  ```
  void inc(Rational this, int x);
  ```

- Zugriffe auf Objektattribute innerhalb einer Objektmethode werden mithilfe dieser Referenz aufgelöst, d.h.:

  ```
  zaehler = zaehler + b * nenner;
  ```

  in der Methode `inc()` ist eigentlich

  ```
  this.zaehler = this.zaehler + b * this.nenner;
  ```

10.2 Klassenattribute

- Objektattribute werden für jedes Objekt neu angelegt,
- Klassenattribute einmal für die gesamte Klasse,
- Klassenattribute erhalten die Qualifizierung `static`

```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8           ...
9 } // end of class Count
```


Die this-Referenz

Woher kommt die Referenz `this`?

- Einem Auruf einer Objektmethode (z.B. `a.inc()`) oder eines Konstruktors wird implizit ein versteckter Parameter übergeben, der auf das Objekt (hier `a`) zeigt.
- Die Signatur von `inc(int x)` ist eigentlich:

  `void inc(Rational this, int x);`
- Zugriffe auf Objektattribute innerhalb einer Objektmethode werden mithilfe dieser Referenz aufgelöst, d.h.:

  `zaehler = zaehler + b * nenner;`

  in der Methode `inc()` ist eigentlich

  `this.zaehler = this.zaehler + b * this.nenner;`

10.2 Klassenattribute

count 0

```
Count a = new Count();
```


10.2 Klassenattribute

- Objektattribute werden für jedes Objekt neu angelegt,
- Klassenattribute einmal für die gesamte Klasse,
- Klassenattribute erhalten die Qualifizierung `static`

```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8           ...
9 } // end of class Count
```

10.2 Klassenattribute

count 0

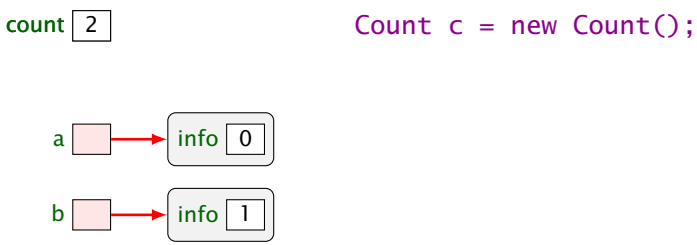
```
Count a = new Count();
```


10.2 Klassenattribute

- Objektattribute werden für jedes Objekt neu angelegt,
- Klassenattribute einmal für die gesamte Klasse,
- Klassenattribute erhalten die Qualifizierung `static`

```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8           ...
9 } // end of class Count
```

10.2 Klassenattribute

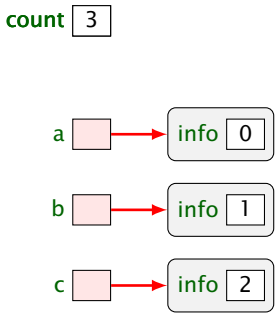


10.2 Klassenattribute

- Objektattribute werden für jedes Objekt neu angelegt,
- Klassenattribute einmal für die gesamte Klasse,
- Klassenattribute erhalten die Qualifizierung `static`

```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8           ...
9 } // end of class Count
```

10.2 Klassenattribute



10.2 Klassenattribute

- Objektattribute werden für jedes Objekt neu angelegt,
- Klassenattribute einmal für die gesamte Klasse,
- Klassenattribute erhalten die Qualifizierung `static`

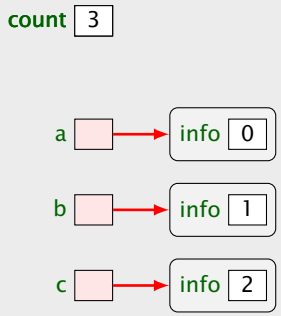
```
1 public class Count {
2     private static int count = 0;
3     private int info;
4     // Konstruktor
5     public Count() {
6         info = count++;
7     }
8           ...
9 } // end of class Count
```

10.2 Klassenattribute

- Das Klassenattribut `count` zählt hier die Anzahl der bereits erzeugten Objekte.
- Das Objektattribut `info` enthält für jedes Objekt eine eindeutige Nummer.
- Außerhalb der Klasse `Class` kann man auf die öffentliche Klassenvariable `name` mit Hilfe von `Class.name` zugreifen.

- Funktionen und Prozeduren der Klasse ohne das implizite `this`-Argument heißen Klassenmethoden und werden auch durch das Schlüsselwort `static` kenntlich gemacht.

10.2 Klassenattribute



Beispiel

In `Rational` könnten wir definieren

```
public static Rational[] intToRationalArray(int[] a) {
    Rational[] b = new Rational[a.length];
    for(int i=0; i < a.length; ++i)
        b[i] = new Rational(a[i]);
    return b;
}
```

- Die Funktion erzeugt für ein Feld von `int`'s ein entsprechendes Feld von `Rational`-Objekten.
- Außerhalb der Klasse `Class` kann die öffentliche Klassenmethode `meth()` mit Hilfe von `Class.meth(...)` aufgerufen werden.

10.2 Klassenattribute

- Das Klassenattribut `count` zählt hier die Anzahl der bereits erzeugten Objekte.
- Das Objektattribut `info` enthält für jedes Objekt eine eindeutige Nummer.
- Außerhalb der Klasse `Class` kann man auf die öffentliche Klassenvariable `name` mit Hilfe von `Class.name` zugreifen.

- Funktionen und Prozeduren der Klasse ohne das implizite `this`-Argument heißen Klassenmethoden und werden auch durch das Schlüsselwort `static` kenntlich gemacht.

Erinnerung:

- Abstrakter Datentyp spezifiziert nur die Operationen
- Implementierung und andere Details sind verborgen

Beispiel

In `Rational` könnten wir definieren

```
public static Rational[] intToRationalArray(int[] a) {
    Rational[] b = new Rational[a.length];
    for(int i=0; i < a.length; ++i)
        b[i] = new Rational(a[i]);
    return b;
}
```

- Die Funktion erzeugt für ein Feld von `int`'s ein entsprechendes Feld von `Rational`-Objekten.
- Außerhalb der Klasse `Class` kann die öffentliche Klassenmethode `meth()` mit Hilfe von `Class.meth(...)` aufgerufen werden.

11 Abstrakte Datentypen

Erinnerung:

- Abstrakter Datentyp spezifiziert nur die Operationen
- Implementierung und andere Details sind verborgen

11.1 Listen

Nachteil von Feldern:

- feste Größe
- Einfügen neuer Elemente nicht möglich
- Streichen ebenfalls nicht

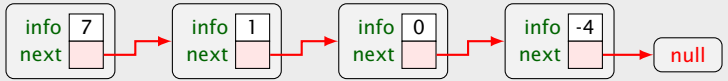
Idee: Listen

11.1 Listen

Nachteil von Feldern:

- feste Größe
- Einfügen neuer Elemente nicht möglich
- Streichen ebenfalls nicht

Idee: Listen



Listen – Version A

`info` : Datenelement der Liste;
`next` : Verweis auf nächstes Element;
`null` : leeres Objekt.

Operationen:

`void insert(int x)` : fügt neues `x` hinter dem aktuellen (ersten) Element ein;

`void delete()` : entfernt Knoten hinter dem aktuellen (ersten) Element;

`String toString()` : liefert eine `String`-Darstellung.

Listen – Version A

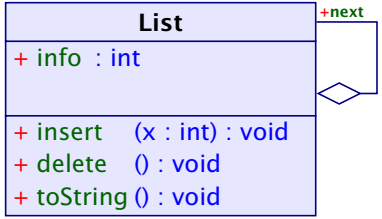
`info` : Datenelement der Liste;
`next` : Verweis auf nächstes Element;
`null` : leeres Objekt.

Operationen:

`void insert(int x)`	:	fügt neues `x` hinter dem aktuellen (ersten) Element ein;
`void delete()`	:	entfernt Knoten hinter dem aktuellen (ersten) Element;
`String toString()`	:	liefert eine `String`-Darstellung.

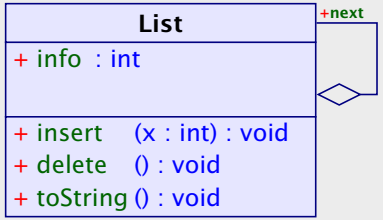
Modellierung

Modellierung als UML-Diagramm:

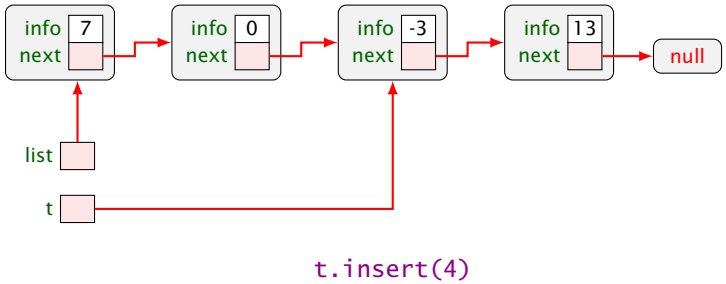


Modellierung

Modellierung als UML-Diagramm:

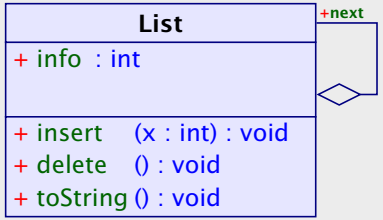


Listen – Insert

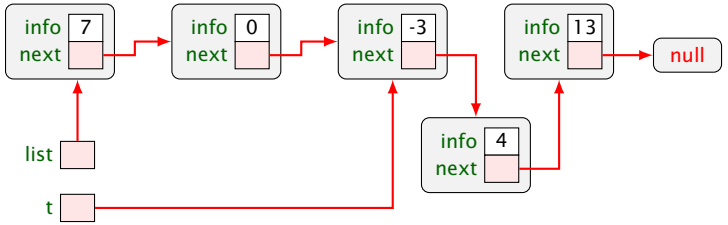


Modellierung

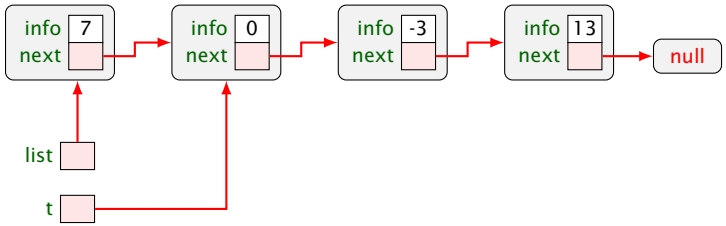
Modellierung als UML-Diagramm:



Listen – Insert

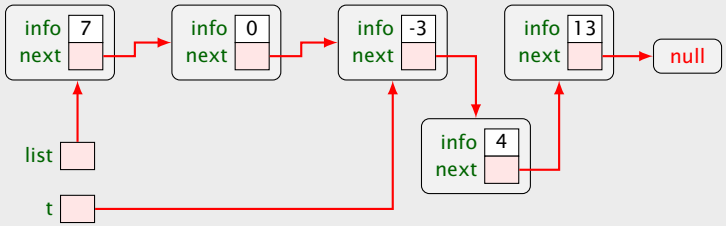


Listen – Delete

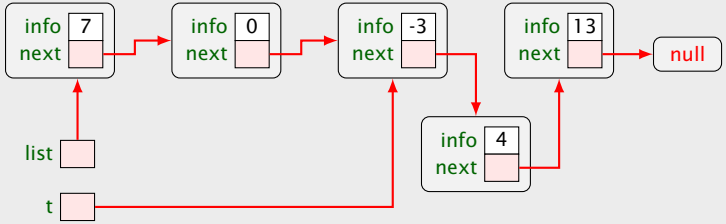



```
t.delete()
```

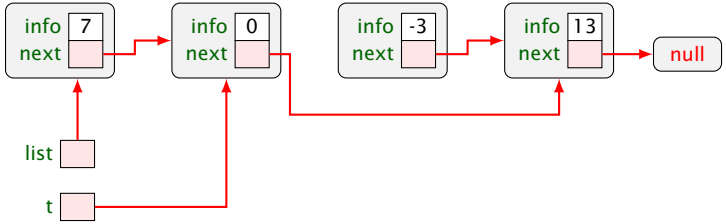

Listen – Insert



Listen – Insert



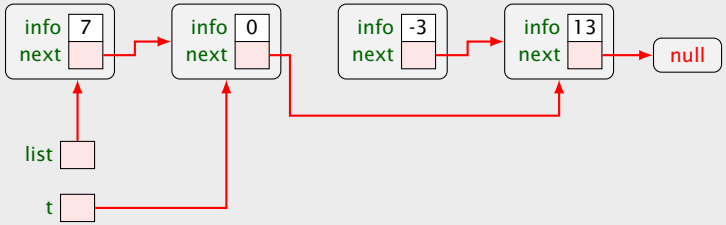
Listen – Delete



Weitere Operationen:

- Liste auf Leerheit testen
- Neue Listen erzeugen ($\Longrightarrow$ Konstruktoren)
  - z.B. eine einelementige Liste
  - eine bestehende Liste verlängern
- Umwandlung zwischen Listen und Feldern...

Listen – Delete



11.1 Listen

Weitere Operationen:

- Liste auf Leerheit testen
- Neue Listen erzeugen ($\Longrightarrow$ Konstruktoren)
  - z.B. eine einelementige Liste
  - eine bestehende Liste verlängern
- Umwandlung zwischen Listen und Feldern...

Listen – Implementierung

```
1  public class List {
2      public int info;
3      public List next;
4
5      // Konstruktoren:
6      public List (int x, List l) {
7          info = x;
8          next = l;
9      }
10     public List (int x) {
11         info = x;
12         next = null;
13     }
14 // continued...
```

Listen – Implementierung

```
15      // Objekt-Methoden:
16      public void insert(int x) {
17          next = new List(x,next);
18      }
19      public void delete() {
20          if (next != null)
21              next = next.next;
22      }
23      public String toString() {
24          String result = "[" + info;
25          for(List t = next; t != null; t = t.next)
26              result = result + ", " + t.info;
27          return result + "]";
28      }
29  // continued...
```


Listen – Implementierung

```
1  public class List {
2      public int info;
3      public List next;
4
5      // Konstruktoren:
6      public List (int x, List l) {
7          info = x;
8          next = l;
9      }
10     public List (int x) {
11         info = x;
12         next = null;
13     }
14 // continued...
```

Listen – Implementierung

```
15      // Objekt-Methoden:
16      public void insert(int x) {
17          next = new List(x,next);
18      }
19      public void delete() {
20          if (next != null)
21              next = next.next;
22      }
23      public String toString() {
24          String result = "[" + info;
25          for(List t = next; t != null; t = t.next)
26              result = result + ", " + t.info;
27          return result + "]";
28      }
29  // continued...
```


Erläuterungen

- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar; sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an, und fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

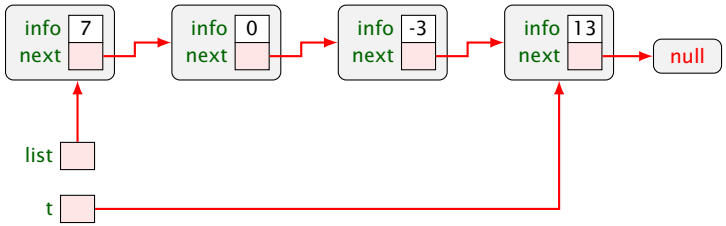
Achtung:
Wenn `delete()` mit dem letzten Listenelement aufgerufen wird, zeigt `next` auf `null`; wir tun dann nichts...

Erläuterungen

- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar; sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an, und fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

Achtung:
Wenn `delete()` mit dem letzten Listenelement aufgerufen wird, zeigt `next` auf `null`; wir tun dann nichts...

Listen – Delete



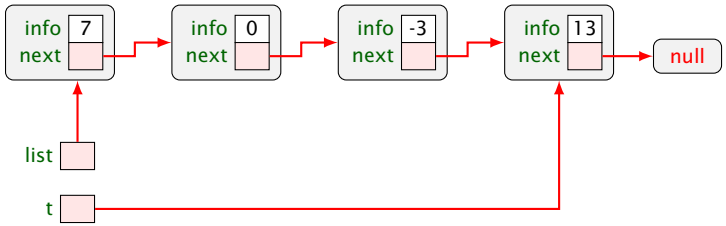
`t.delete()`

Erläuterungen

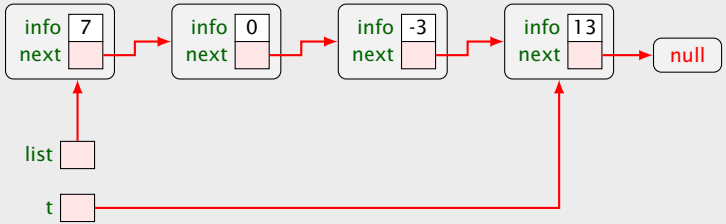
- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar; sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an, und fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

Achtung:
Wenn `delete()` mit dem letzten Listenelement aufgerufen wird, zeigt `next` auf `null`; wir tun dann nichts...

Listen – Delete



Listen – Delete



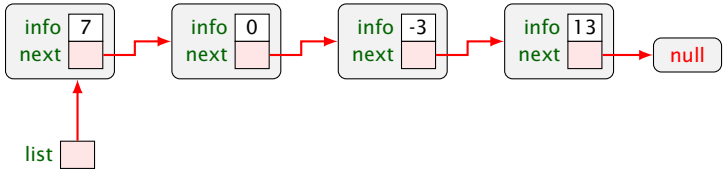
Erläuterungen

Weil Objektmethoden nur für von `null` verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels `toString()` als `String` dargestellt werden.

Der Konkatenations-Operator `+` ist so schlau, vor Aufruf von `toString()` zu überprüfen, ob ein `null`-Objekt vorliegt. Ist das der Fall, wird "null" ausgegeben.

Für eine andere Darstellung benötigen wir eine Klassenmethode `toString(List l)`;

Listen – toString()




```
write("" + list);
```

liefert: „[7, 0, -3, 13]“

Erläuterungen

Weil Objektmethoden nur für von `null` verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels `toString()` als `String` dargestellt werden.

Der Konkatenations-Operator `+` ist so schlau, vor Aufruf von `toString()` zu überprüfen, ob ein `null`-Objekt vorliegt. Ist das der Fall, wird "null" ausgegeben.

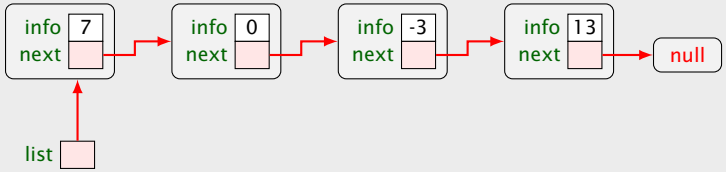
Für eine andere Darstellung benötigen wir eine Klassenmethode `toString(List l)`;

Listen – toString()


```
write("" + list);
```

liefert: „null“

Listen – toString()




```
write("" + list);
```

liefert: „[7, 0, -3, 13]“

Listen – Implementierung

```
30      // Klassen-Methoden:
31      public static boolean isEmpty(List l) {
32          return (l == null);
33      }
34      public static String toString(List l) {
35          if (l == null)
36              return "[]";
37          else
38              return l.toString();
39      }
40  // continued...
```


Listen – toString()


```
write("" + list);
```

liefert: „null“

Listen – Implementierung

```
30      // Klassen-Methoden:
31      public static boolean isEmpty(List l) {
32          return (l == null);
33      }
34      public static String toString(List l) {
35          if (l == null)
36              return "[]";
37          else
38              return l.toString();
39      }
40  // continued...
```


Listen – Implementierung

```
41      public static List arrayToList(int[] a) {
42          List result = null;
43          for(int i = a.length-1; i >= 0; --i)
44              result = new List(a[i], result);
45          return result;
46      }
47      public int[] listToArray() {
48          List t  = this;
49          int n   = length();
50          int[] a = new int[n];
51          for(int i = 0; i < n; ++i) {
52              a[i] = t.info;
53              t = t.next;
54          }
55          return a;
56      }
57  // continued...
```

Listen – Implementierung

- Damit das erste Element der Ergebnisliste `a[0]` enthält, beginnt die Iteration in `arrayToList()` beim größten Element.
- `listToArray()` ist als Objekt-Methode realisiert und funktioniert darum nur für nicht-leere Listen.
- Wir benötigen die Länge einer Liste:

```
58     private int length() {
59         int result = 1;
60         for(List t = next; t != null; t = t.next)
61             result++;
62         return result;
63     }
64 } // end of class List
```


Listen – Implementierung

```
41     public static List arrayToList(int[] a) {
42         List result = null;
43         for(int i = a.length-1; i >= 0; --i)
44             result = new List(a[i], result);
45         return result;
46     }
47     public int[] listToArray() {
48         List t  = this;
49         int n   = length();
50         int[] a = new int[n];
51         for(int i = 0; i < n; ++i) {
52             a[i] = t.info;
53             t = t.next;
54         }
55         return a;
56     }
57 // continued...
```

Listen – Implementierung

- Damit das erste Element der Ergebnisliste `a[0]` enthält, beginnt die Iteration in `arrayToList()` beim größten Element.
- `listToArray()` ist als Objekt-Methode realisiert und funktioniert darum nur für nicht-leere Listen.
- Wir benötigen die Länge einer Liste:

```
58     private int length() {
59         int result = 1;
60         for(List t = next; t != null; t = t.next)
61             result++;
62         return result;
63     }
64 } // end of class List
```


Listen – Implementierung

- Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- Damit `length()` auch für `null` funktioniert, hätten wir analog zu `toString()` auch noch eine Klassen-Methode `int length(List l)` definieren können.
- Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode `static int[] listToArray (List l)` zu definieren, die auch für leere Listen definiert ist.

Listen – Implementierung

- Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- Damit `length()` auch für `null` funktioniert, hätten wir analog zu `toString()` auch noch eine Klassen-Methode `int length(List l)` definieren können.
- Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode `static int[] listToArray (List l)` zu definieren, die auch für leere Listen definiert ist.

Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der auf auf der Mischoperation beruht.

John von Neumann (1945)

Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der auf auf der Mischoperation beruht.

John von Neumann (1945)

Mergesort – Sortieren durch Mischen

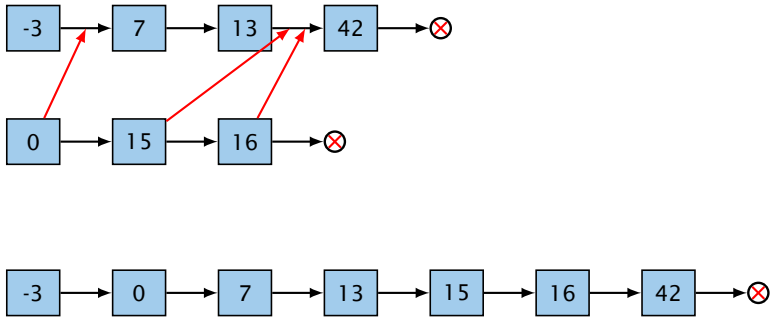
Die Mischoperation

Input: zwei sortierte Listen

Output: eine gemeinsame sortierte Liste

Später bauen wir damit einen Sortieralgorithmus...

Beispiel – Mischen



Mergesort – Sortieren durch Mischen

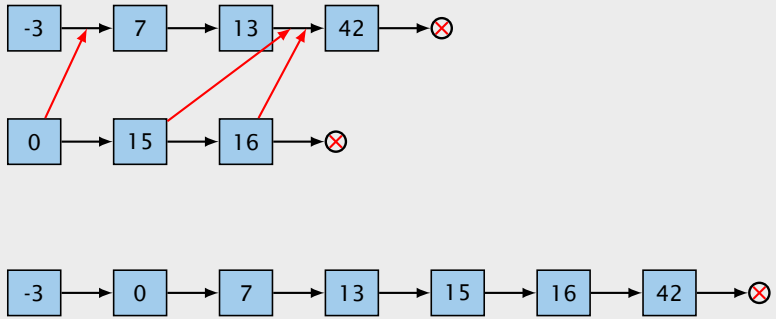
Die Mischoperation

Input: zwei sortierte Listen

Output: eine gemeinsame sortierte Liste

Später bauen wir damit einen Sortieralgorithmus...

Beispiel – Mischen



Mergesort – Sortieren durch Mischen

Idee:

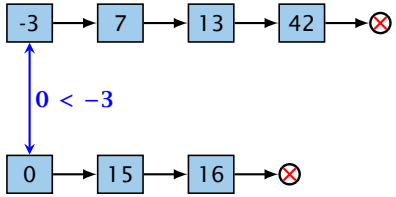
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

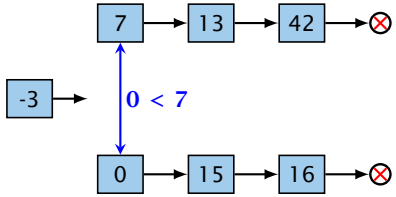


Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

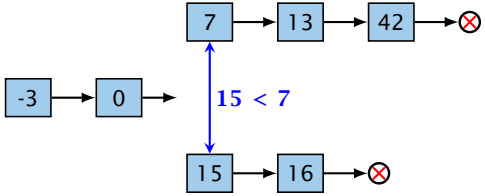


Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

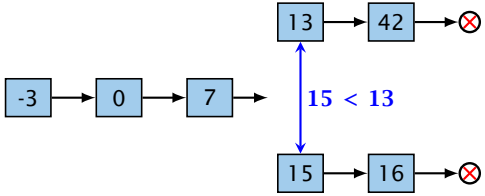


Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

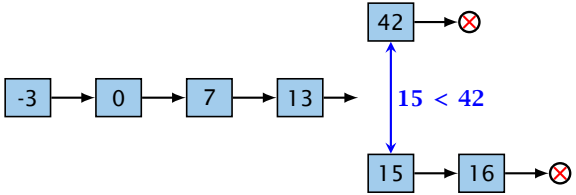


Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

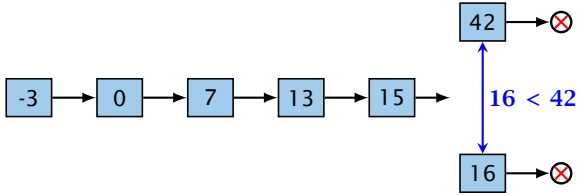


Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

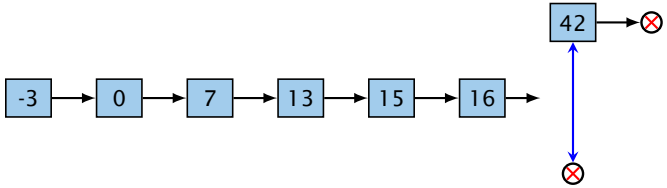


Mergesort – Sortieren durch Mischen

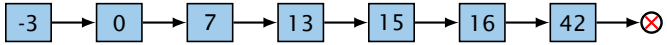
Idee:

- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen



Beispiel – Mischen



Mergesort – Sortieren durch Mischen

Idee:

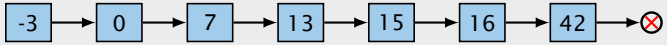
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Mergesort – Implementierung

Rekursive Implementierung

- Falls eine der beiden Listen a und b leer ist, geben wir die andere aus.
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten. . .

Beispiel – Mischen



Mergesort – Implementierung

```
1  public static List merge(List a, List b) {
2      if (b == null)
3          return a;
4      if (a == null)
5          return b;
6      if (b.info > a.info) {
7          a.next = merge(a.next, b);
8          return a;
9      } else {
10         b.next = merge(a, b.next);
11         return b;
12     }
13 }
```


Mergesort – Implementierung

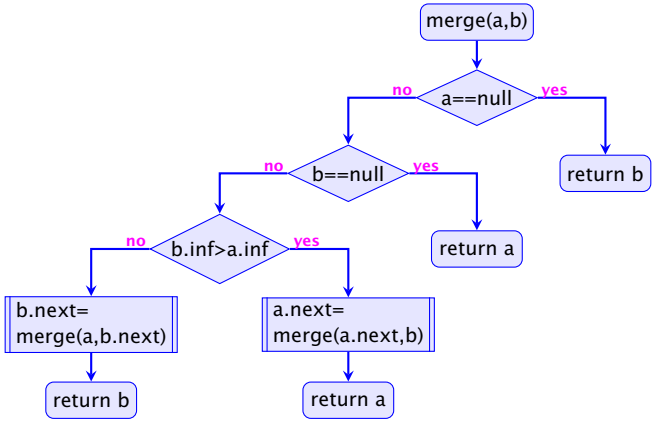
Rekursive Implementierung

- Falls eine der beiden Listen a und b leer ist, geben wir die andere aus.
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten. . .

Mergesort – Implementierung

```
1  public static List merge(List a, List b) {
2      if (b == null)
3          return a;
4      if (a == null)
5          return b;
6      if (b.info > a.info) {
7          a.next = merge(a.next, b);
8          return a;
9      } else {
10         b.next = merge(a, b.next);
11         return b;
12     }
13 }
```


Kontrollfluss

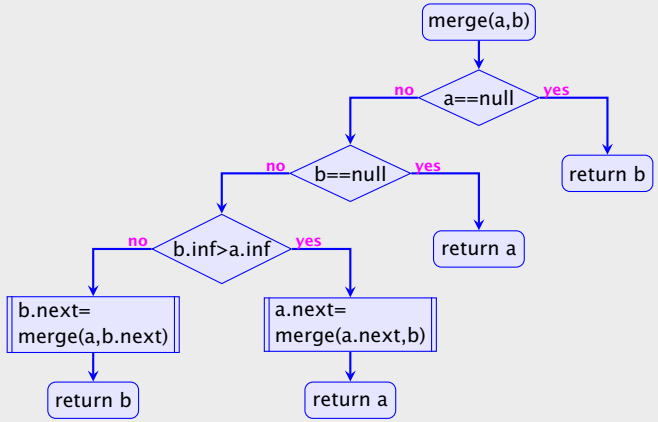


Mergesort

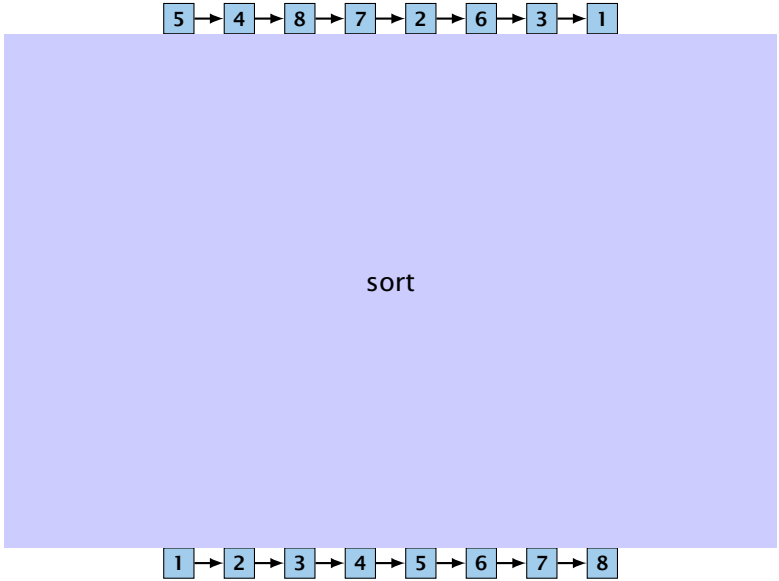
Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

Kontrollfluss



Mergesort

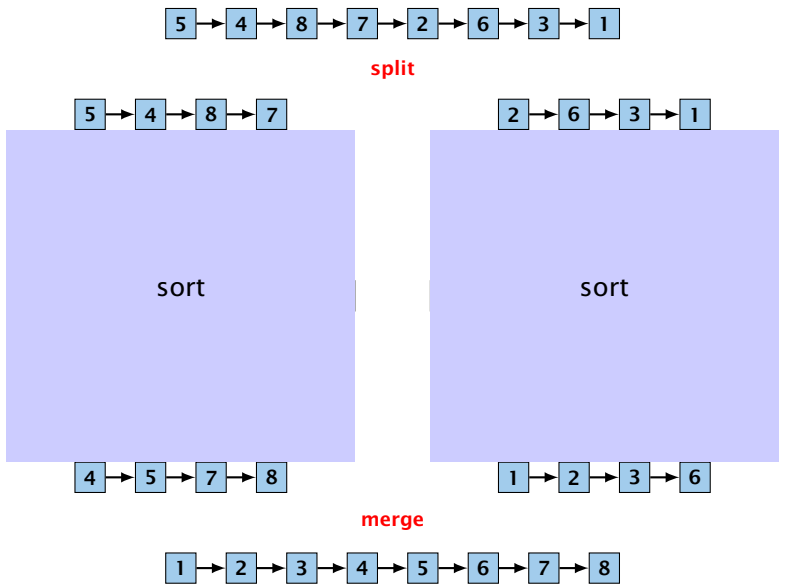


Mergesort

Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

Mergesort

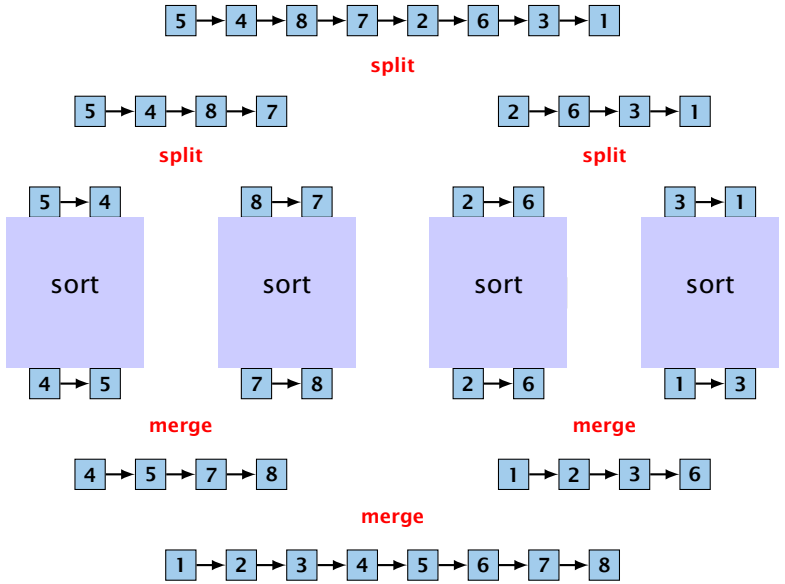


Mergesort

Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

Mergesort

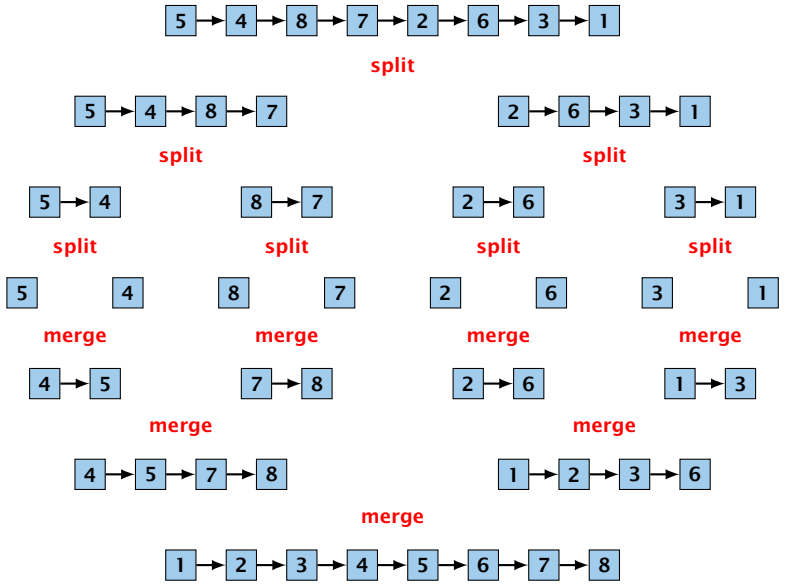


Mergesort

Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

Mergesort



Mergesort

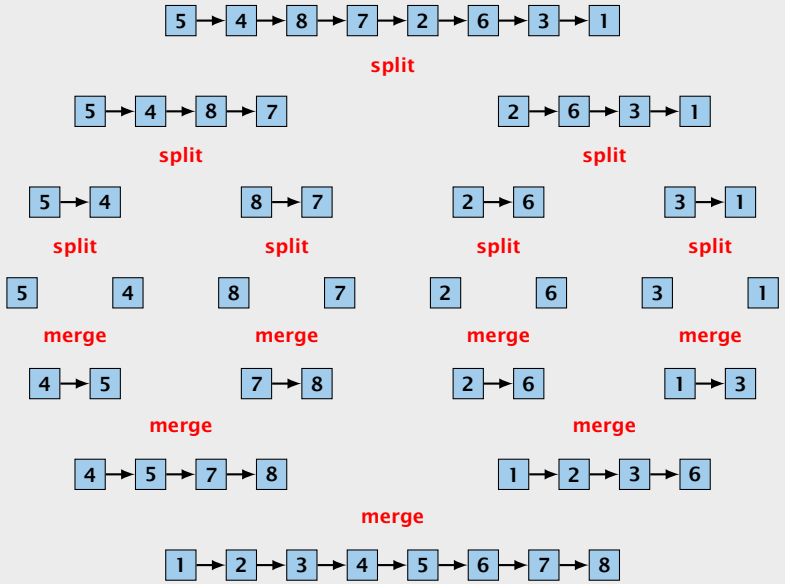
Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

Mergesort – Implementierung

```
1 public static List sort(List a) {
2     if (a == null || a.next == null)
3         return a;
4     List b = a.half(); // Halbiere!
5     a = sort(a);
6     b = sort(b);
7     return merge(a,b);
8 }
```


Mergesort



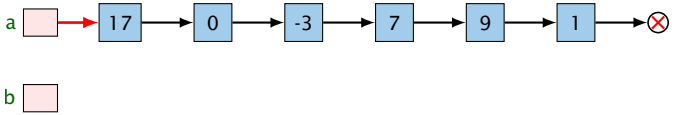
Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```


Mergesort – Implementierung

```
1 public static List sort(List a) {
2     if (a == null || a.next == null)
3         return a;
4     List b = a.half(); // Halbiere!
5     a = sort(a);
6     b = sort(b);
7     return merge(a,b);
8 }
```

Halbieren

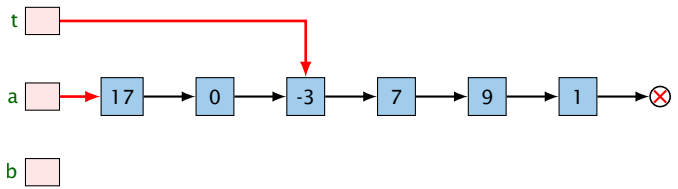


a.half()

Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```

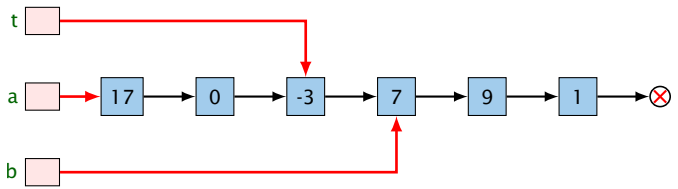
Halbieren



Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```

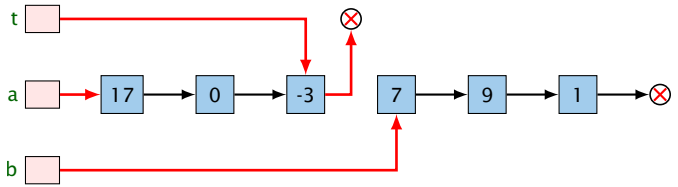
Halbieren



Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```

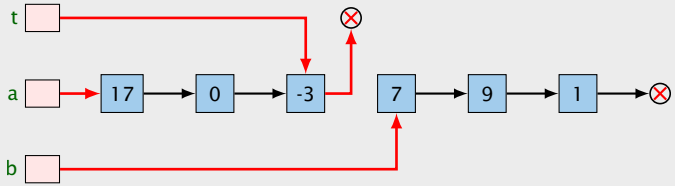
Halbieren



Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```

Halbieren



Mergesort – Analyse

- Sei $V(n)$ die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge n benötigt.
  Dann gilt:

$$\begin{aligned} V(1) &= 0 \\ V(2n) &\leq 2 \cdot V(n) + 2 \cdot n \end{aligned}$$

- Für $n = 2^k$, sind das dann nur $k \cdot n = n \log_2 n$ Vergleiche!!!

Mergesort – Analyse

- Sei $V(n)$ die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge n benötigt.
  Dann gilt:

$$\begin{aligned} V(1) &= 0 \\ V(2n) &\leq 2 \cdot V(n) + 2 \cdot n \end{aligned}$$

- Für $n = 2^k$, sind das dann nur $k \cdot n = n \log_2 n$ Vergleiche!!!

Mergesort – Bemerkungen

Achtung:

- Unsere Funktion `sort()` zerstört ihr Argument!
- Alle Listenknoten der Eingabe werden weiterverwendet.
- Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden (wie?)
- Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ implementieren (wie?)

11.2 Keller (Stacks)

Operationen:

`boolean isEmpty()`	:	testet auf Leerheit;
`int pop()`	:	liefert oberstes Element;
`void push(int x)`	:	legt `x` oben auf dem Keller ab;
`String toString()`	:	liefert eine String-Darstellung

Weiterhin müssen wir einen leeren Keller anlegen können.

Mergesort – Bemerkungen

Achtung:

- Unsere Funktion `sort()` zerstört ihr Argument!
- Alle Listenknoten der Eingabe werden weiterverwendet.
- Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden (wie?)
- Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ implementieren (wie?)

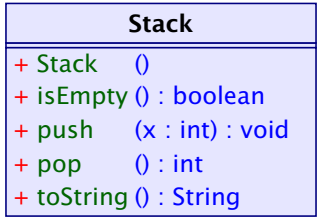
11.2 Keller (Stacks)

Operationen:

`boolean isEmpty()`	:	testet auf Leerheit;
`int pop()`	:	liefert oberstes Element;
`void push(int x)`	:	legt `x` oben auf dem Keller ab;
`String toString()`	:	liefert eine String-Darstellung

Weiterhin müssen wir einen leeren Keller anlegen können.

Modellierung Stack



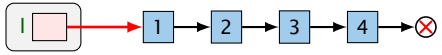
Modellierung Stack

Stack
+ Stack ()
+ isEmpty () : boolean
+ push (x : int) : void
+ pop () : int
+ toString () : String

Stack via List

Idee

- Realisiere Stack mithilfe einer Liste:

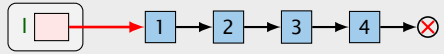


- Das Attribut l zeigt auf das oberste Kellerelement.

Stack via List

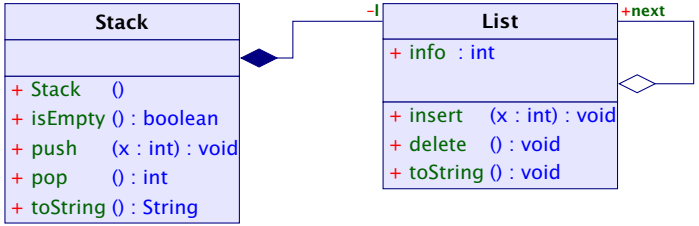
Idee

- Realisiere Stack mithilfe einer Liste:



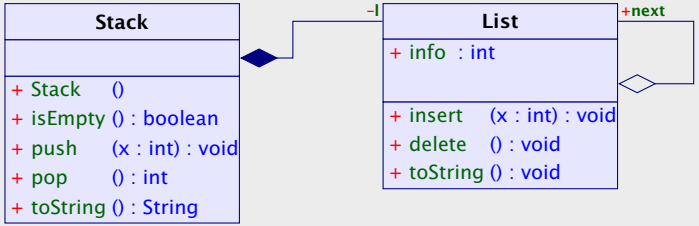
- Das Attribut l zeigt auf das oberste Kellerelement.

Modellierung Stack via List



Die gefüllte Raute bezeichnet eine Komposition. Die Liste existiert nur solange wie der Stack (d.h. wird üblicherweise durch diesen erzeugt und zerstört). Außerdem kann die Liste nur Teil eines Stacks sein.

Modellierung Stack via List



Die gefüllte Raute bezeichnet eine Komposition. Die Liste existiert nur solange wie der Stack (d.h. wird üblicherweise durch diesen erzeugt und zerstört). Außerdem kann die Liste nur Teil eines Stacks sein.

Stack – Implementierung

```
1  public class Stack {
2      private List l;
3      // Konstruktor :
4      public Stack() {
5          l = null;
6      }
7      // Objektmethoden :
8      public boolean isEmpty() {
9          return l == null;
10     }
11     // continued...
```

Stack – Implementierung

```
12      public int pop() {
13          int result = l.info;
14          l = l.next;
15          return result;
16      }
17      public void push(int a) {
18          l = new List(a,l);
19      }
20      public String toString() {
21          return List.toString(l);
22      }
23  } // end of class Stack
```


Stack – Implementierung

```
1  public class Stack {
2      private List l;
3      // Konstruktor :
4      public Stack() {
5          l = null;
6      }
7      // Objektmethoden :
8      public boolean isEmpty() {
9          return l == null;
10     }
11     // continued...
```

Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Stack – Implementierung

```
12      public int pop() {
13          int result = l.info;
14          l = l.next;
15          return result;
16      }
17      public void push(int a) {
18          l = new List(a,l);
19      }
20      public String toString() {
21          return List.toString(l);
22      }
23  } // end of class Stack
```

Stack – Implementierung

```
12      public int pop() {
13          int result = l.info;
14          l = l.next;
15          return result;
16      }
17      public void push(int a) {
18          l = new List(a,l);
19      }
20      public String toString() {
21          return List.toString(l);
22      }
23  } // end of class Stack
```

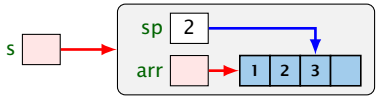

Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Stack via Array



`s.push(4)`

Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

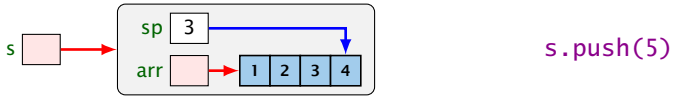
Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Stack via Array



`s.push(5)`

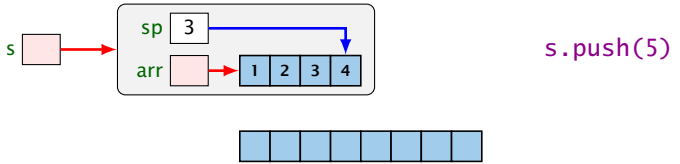
Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Stack via Array



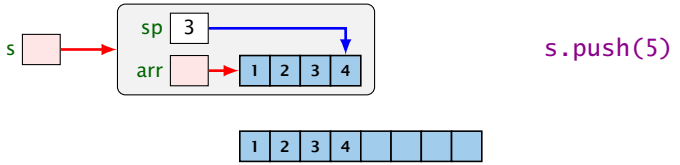
Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

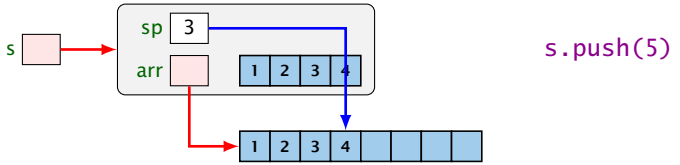
- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Stack via Array



`s.push(5)`

Stack via Array



`s.push(5)`

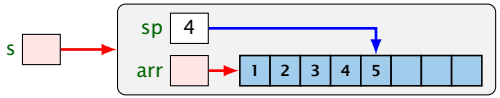
Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Stack via Array



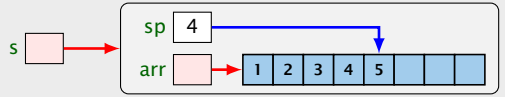
Bemerkungen

- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

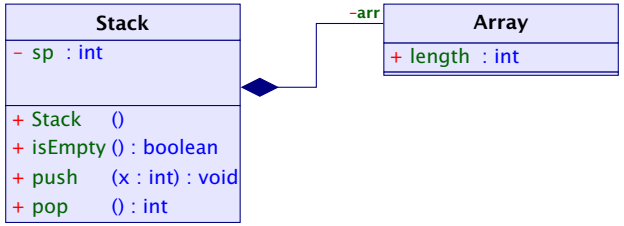
Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über ersetzen wir es durch ein größeres.

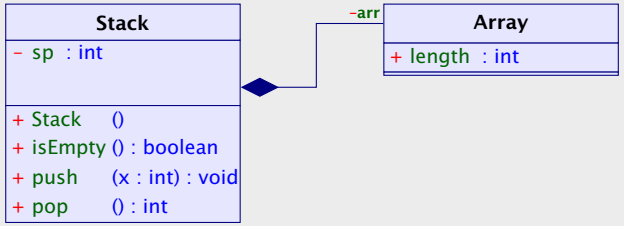
Stack via Array



Modellierung Stack



Modellierung Stack



Implementierung

```
1  public class Stack {
2      private int sp;
3      private int[] arr;
4      // Konstruktoren:
5      public Stack() {
6          sp  = -1;
7          arr = new int[4];
8      }
9      // Objekt-Methoden:
10     public boolean isEmpty() {
11         return sp < 0;
12     }
13     // continued...
```

Implementierung

```
14      public int pop() {
15          return arr[sp--];
16      }
17      public void push(int x) {
18          ++sp;
19          if (sp == arr.length) {
20              int[] b = new int[2*sp];
21              for (int i = 0; i < sp; ++i) b[i] = arr[i];
22              arr = b;
23          }
24          arr[sp] = x;
25      }
26      public toString() {...}
27  } // end of class Stack
```


Implementierung

```
1  public class Stack {
2      private int sp;
3      private int[] arr;
4      // Konstruktoren:
5      public Stack() {
6          sp  = -1;
7          arr = new int[4];
8      }
9      // Objekt-Methoden:
10     public boolean isEmpty() {
11         return sp < 0;
12     }
13     // continued...
```

Implementierung

```
14      public int pop() {
15          return arr[sp--];
16      }
17      public void push(int x) {
18          ++sp;
19          if (sp == arr.length) {
20              int[] b = new int[2*sp];
21              for (int i = 0; i < sp; ++i) b[i] = arr[i];
22              arr = b;
23          }
24          arr[sp] = x;
25      }
26      public toString() {...}
27  } // end of class Stack
```


11.2 Keller (Stacks)

Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

11.2 Keller (Stacks)

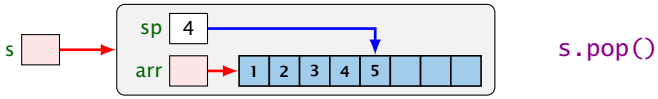
Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

Stack via Array




```
s.pop()
```

11.2 Keller (Stacks)

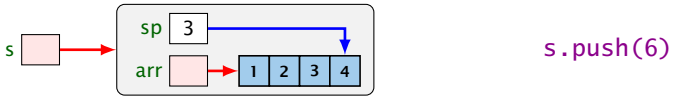
Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

Stack via Array




```
s.push(6)
```

11.2 Keller (Stacks)

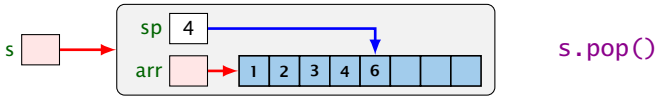
Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

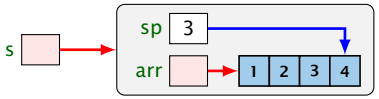
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

Stack via Array




```
s.pop()
```

Stack via Array



`s.push(7)`

11.2 Keller (Stacks)

Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

11.2 Keller (Stacks)

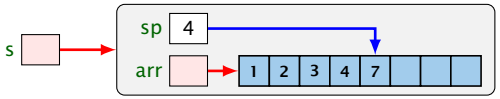
Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

Stack via Array



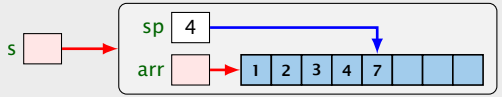
11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array



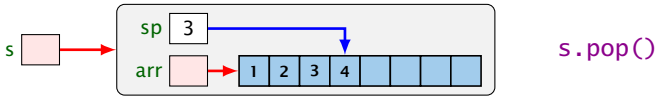
11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array




```
s.pop()
```

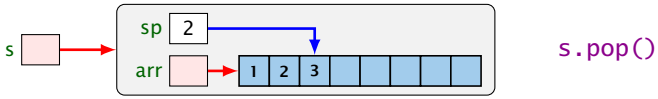
11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

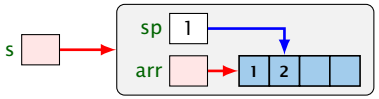
- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array




```
s.pop()
```

Stack via Array




```
s.push(5)
```


11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

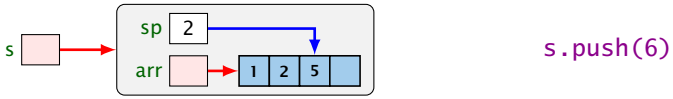
11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array



`s.push(6)`

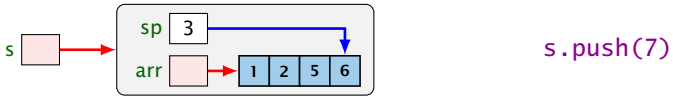
11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array



`s.push(7)`

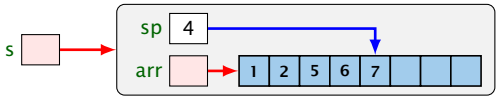
11.2 Keller (Stacks)

- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

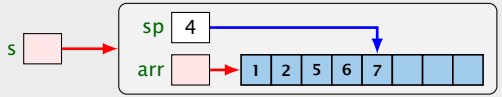
Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array



Stack via Array



11.2 Keller (Stacks)

Beobachtung:

- Vor jedem Kopieren werden mindestens halb so viele Operationen ausgeführt, wie Elemente kopiert werden.
- Gemittelt über die gesamte Folge der Operationen werden pro Operation maximal zwei Zahlen kopiert (↑amortisierte Aufwandsanalyse)

11.2 Keller (Stacks)

Beobachtung:

- Vor jedem Kopieren werden mindestens halb so viele Operationen ausgeführt, wie Elemente kopiert werden.
- Gemittelt über die gesamte Folge der Operationen werden pro Operation maximal zwei Zahlen kopiert (↑amortisierte Aufwandsanalyse)

Implementierung

```
1  public int pop() {
2      int result = arr[sp];
3      if (sp == arr.length/4 && sp >= 2) {
4          int[] b = new int[2*sp];
5          for(int i = 0; i < sp; ++i)
6              b[i] = arr[i];
7          arr = b;
8      }
9      sp--;
10     return result;
11 }
```

Implementierung

```
1  public int pop() {
2      int result = arr[sp];
3      if (sp == arr.length/4 && sp >= 2) {
4          int[] b = new int[2*sp];
5          for(int i = 0; i < sp; ++i)
6              b[i] = arr[i];
7          arr = b;
8      }
9      sp--;
10     return result;
11 }
```


11.3 Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem FIFO-Prinzip (First-In-First-Out).

Operationen:

`boolean isEmpty()`	:	testet auf Leerheit;
`int dequeue()`	:	liefert erstes Element;
`void enqueue(int x)`	:	reiht x in die Schlange ein;
`String toString()`	:	liefert eine String-Darstellung.

Weiterhin müssen wir eine leere Schlange anlegen können.

11.3 Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem FIFO-Prinzip (First-In-First-Out).

Operationen:

`boolean isEmpty()`	:	testet auf Leerheit;
`int dequeue()`	:	liefert erstes Element;
`void enqueue(int x)`	:	reiht x in die Schlange ein;
`String toString()`	:	liefert eine String-Darstellung.

Weiterhin müssen wir eine leere Schlange anlegen können.

Modellierung Queue

Queue
+ Queue ()
+ isEmpty () : boolean
+ enqueue (x : int) : void
+ dequeue () : int
+ toString () : String

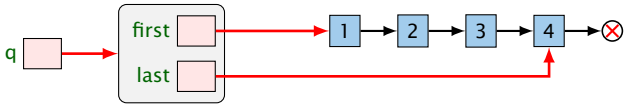
Modellierung Queue

Queue
+ Queue ()
+ isEmpty () : boolean
+ enqueue (x : int) : void
+ dequeue () : int
+ toString () : String

Queue via List

Erste Idee:

- Realisiere Schlange mithilfe einer Liste:

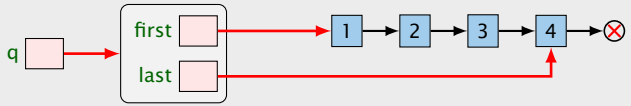


- `first` zeigt auf das nächste zu entnehmende Element;
- `last` zeigt auf das Element hinter dem eingefügt wird;

Queue via List

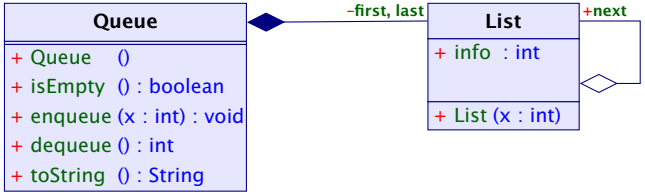
Erste Idee:

- Realisiere Schlange mithilfe einer Liste:



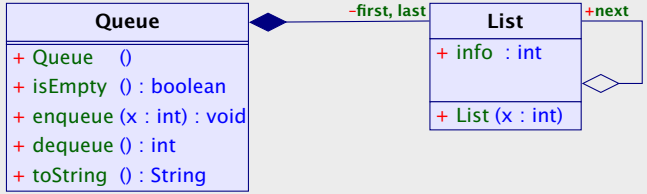
- `first` zeigt auf das nächste zu entnehmende Element;
- `last` zeigt auf das Element hinter dem eingefügt wird;

Modellierung: Queue via List



Objekte der Klasse `Queue` enthalten zwei Verweise auf Objekte der Klasse `List`.

Modellierung: Queue via List



Objekte der Klasse Queue enthalten zwei Verweise auf Objekte der Klasse List.

Queue – Implementierung

```
1  public class Queue {
2      private List first, last;
3      // Konstruktor:
4      public Queue() {
5          first = last = null;
6      }
7      // Objekt-Methoden:
8      public boolean isEmpty() {
9          return List.isEmpty(first);
10     }
11     // continued...
```

Queue – Implementierung

```
12      public int dequeue() {
13          int result = first.info;
14          if (last == first) last = null;
15          first = first.next;
16          return result;
17      }
18      public void enqueue(int x) {
19          if (first == null)
20              first = last = new List(x);
21          else {
22              last.insert(x);
23              last = last.next;
24          }
25      }
26      public String toString() {
27          return List.toString(first);
28      }
29  } // end of class Queue
```

Queue – Implementierung

```
1  public class Queue {
2      private List first, last;
3      // Konstruktor:
4      public Queue() {
5          first = last = null;
6      }
7      // Objekt-Methoden:
8      public boolean isEmpty() {
9          return List.isEmpty(first);
10     }
11     // continued...
```

Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue – Implementierung

```
12      public int dequeue() {
13          int result = first.info;
14          if (last == first) last = null;
15          first = first.next;
16          return result;
17      }
18      public void enqueue(int x) {
19          if (first == null)
20              first = last = new List(x);
21          else {
22              last.insert(x);
23              last = last.next;
24          }
25      }
26      public String toString() {
27          return List.toString(first);
28      }
29  } // end of class Queue
```

Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

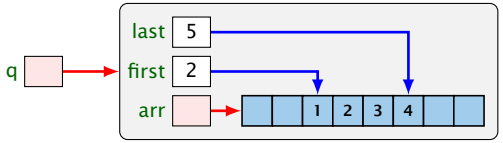
Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue – Implementierung

```
12      public int dequeue() {
13          int result = first.info;
14          if (last == first) last = null;
15          first = first.next;
16          return result;
17      }
18      public void enqueue(int x) {
19          if (first == null)
20              first = last = new List(x);
21          else {
22              last.insert(x);
23              last = last.next;
24          }
25      }
26      public String toString() {
27          return List.toString(first);
28      }
29  } // end of class Queue
```

Queue via Array



`q.dequeue()`

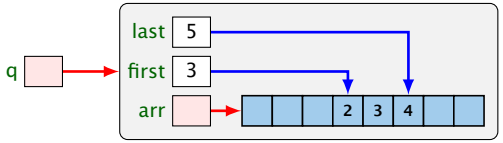
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



`q.enqueue(7)`

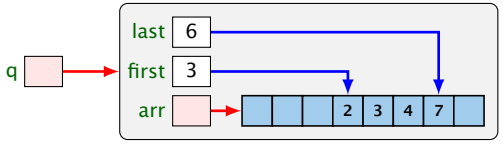
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



`q.enqueue(9)`

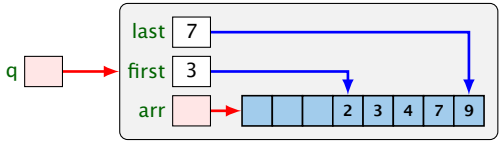
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



`q.enqueue(0)`

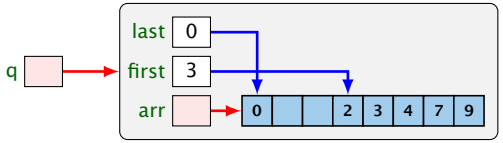
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



`q.enqueue(4)`

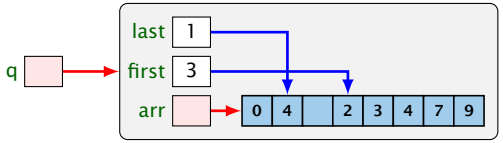
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



`q.enqueue(3)`

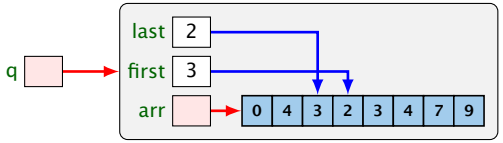
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



`q.enqueue(8)`

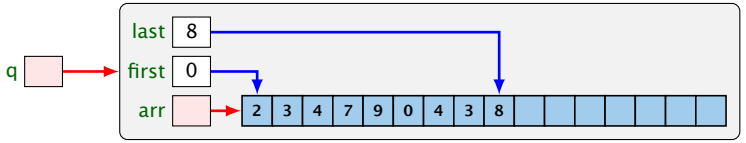
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

Queue via Array



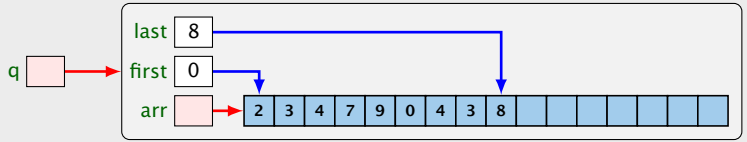
Bemerkungen

- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

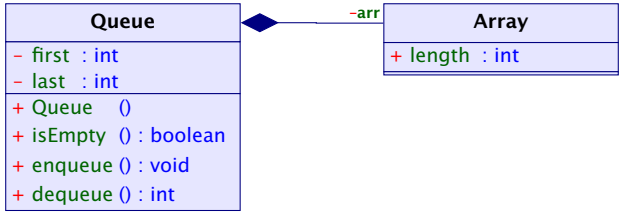
Zweite Idee:

- Realisiere Keller mithilfe eines Feldes, und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über ersetzen wir es durch ein größeres.

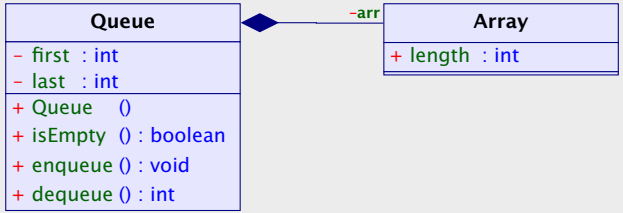
Queue via Array



Modellierung: Queue via Array



Modellierung: Queue via Array



Implementierung

```
1  public class Queue {
2      private int first, last;
3      private int[] arr;
4      // Konstruktor:
5      public Queue() {
6          first = last = -1;
7          arr = new int[4];
8      }
9      // Objekt-Methoden:
10     public boolean isEmpty() { return first == -1; }
11     public String toString() {...}
12     //continued...
```

Implementierung

```
1  public class Queue {
2      private int first, last;
3      private int[] arr;
4      // Konstruktor:
5      public Queue() {
6          first = last = -1;
7          arr = new int[4];
8      }
9      // Objekt-Methoden:
10     public boolean isEmpty() { return first == -1; }
11     public String toString() {...}
12     //continued...
```


Implementierung von enqueue()

- Falls die Schlange leer war, muss `first` und `last` auf `0` gesetzt werden.
- Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- In diesem Fall legen wir ein Feld doppelter Größe an.

  Die Elemente `a[first],..., a[a.length-1]`, `a[0], a[1],..., a[first-1]` kopieren wir nach `b[0],..., b[a.length-1]`.
- Dann setzen wir `first = 0; last = a.length; a = b;`
- Nun kann `x` an der Stelle `a[last]` abgelegt werden.

Implementierung

```
13    public void enqueue(int x) {
14        if (first == -1) {
15            first = last = 0;
16        } else {
17            int n = arr.length;
18            last = (last + 1) % n;
19            if (last == first) {
20                int[] b = new int[2*n];
21                for (int i = 0; i < n; ++i)
22                    b[i] = arr[(first + i) % n];
23                first = 0;
24                last  = n;
25                arr   = b;
26            }
27        } // end if and else
28        arr[last] = x;
29    }
```


Implementierung von enqueue()

- Falls die Schlange leer war, muss `first` und `last` auf `0` gesetzt werden.
- Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- In diesem Fall legen wir ein Feld doppelter Größe an.

  Die Elemente `a[first],..., a[a.length-1]`, `a[0], a[1],..., a[first-1]` kopieren wir nach `b[0],..., b[a.length-1]`.
- Dann setzen wir `first = 0; last = a.length; a = b;`
- Nun kann `x` an der Stelle `a[last]` abgelegt werden.

Implementierung

```
13      public void enqueue(int x) {
14          if (first == -1) {
15              first = last = 0;
16          } else {
17              int n = arr.length;
18              last = (last + 1) % n;
19              if (last == first) {
20                  int[] b = new int[2*n];
21                  for (int i = 0; i < n; ++i)
22                      b[i] = arr[(first + i) % n];
23                  first = 0;
24                  last  = n;
25                  arr   = b;
26              }
27          } // end if and else
28          arr[last] = x;
29      }
```


Implementierung von dequeue()

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden first und last auf `-1` gesetzt.
- Andernfalls wird first um `1` (modulo der Länge von `arr`) inkrementiert.

Für eine evt. Freigabe unterscheiden wir zwei Fälle.

1. Ist `first < last`, liegen die Schlangen-Elemente an den Stellen `arr[first]`, ..., `arr[last]`. Sind dies höchstens $n/4$, werden sie an die Stellen `b[0]`, ..., `b[last-first]` kopiert.

Implementierung

```
13 public int dequeue() {
14     int result = a[first];
15     if (last == first) {
16         first = last = -1;
17         return result;
18     }
19     int n = arr.length;
20     first = (first+1) % n;
21     int diff = last - first;
22     if (diff > 0 && diff < n/4) {
23         int[] b = new int[n/2];
24         for (int i = first; i <= last; ++i)
25             b[i-first] = a[i];
26         last = last - first;
27         first = 0;
28         arr = b;
29     } else // continued...
```


Implementierung von dequeue()

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden first und last auf `-1` gesetzt.
- Andernfalls wird first um `1` (modulo der Länge von `arr`) inkrementiert.

Für eine evt. Freigabe unterscheiden wir zwei Fälle.

1. Ist `first < last`, liegen die Schlangen-Elemente an den Stellen `arr[first]`, ..., `arr[last]`. Sind dies höchstens $n/4$, werden sie an die Stellen `b[0]`, ..., `b[last-first]` kopiert.

Implementierung

```
13 public int dequeue() {
14     int result = a[first];
15     if (last == first) {
16         first = last = -1;
17         return result;
18     }
19     int n = arr.length;
20     first = (first+1) % n;
21     int diff = last - first;
22     if (diff > 0 && diff < n/4) {
23         int[] b = new int[n/2];
24         for (int i = first; i <= last; ++i)
25             b[i-first] = a[i];
26         last = last - first;
27         first = 0;
28         arr = b;
29     } else // continued...
```


Implementierung von dequeue()

2. Ist `last < first`, liegen die Schlangen-Elemente an den Stellen `arr[0]`, ..., `arr[last]` und `arr[first]`, ..., `arr[arr.length-1]`.

   Sind dies höchstens $n/4$, werden sie an die Stellen `b[0]`, ..., `b[last]` sowie `b[first-n/2]`, ..., `b[n/2-1]` kopiert.

   - `first` und `last` müssen die richtigen neuen Werte erhalten.
   - Dann kann `a` durch `b` ersetzt werden.

Implementierung

```
22      if (diff < 0 && diff + n < n/4) {
23          int[] b = new int[n/2];
24          for (int i = 0; i <= last; ++i)
25              b[i] = arr[i];
26          for (int i = first; i < n; ++i)
27              b[i-n/2] = arr[i];
28          first = first-n/2;
29          arr = b;
30      }
31      return result;
32  }
```


Implementierung von dequeue()

2. Ist `last < first`, liegen die Schlangen-Elemente an den Stellen `arr[0]`, ..., `arr[last]` und `arr[first]`, ..., `arr[arr.length-1]`.

   Sind dies höchstens $n/4$, werden sie an die Stellen `b[0]`, ..., `b[last]` sowie `b[first-n/2]`, ..., `b[n/2-1]` kopiert.

   - `first` und `last` müssen die richtigen neuen Werte erhalten.
   - Dann kann `a` durch `b` ersetzt werden.

Zusammenfassung

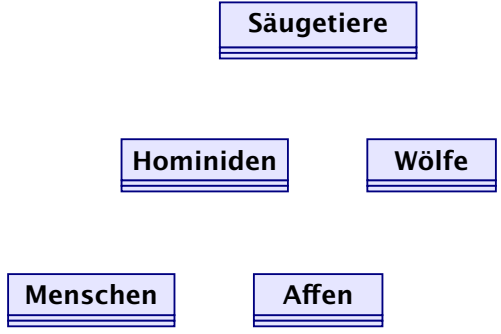
- Der Datentyp `List` ist nicht sehr abstract, dafür extrem flexibel (gut für rapid prototyping)
- Für die nützlichen (eher) abstrakten Datentypen `Stack` und `Queue` lieferten wir zwei Implementierungen. Einer sehr einfache, und eine cache-effiziente.
- **Achtung:** oft werden bei diesen Datentypen noch weitere Operationen zur Verfügung gestellt.

Implementierung

```
22      if (diff < 0 && diff + n < n/4) {
23          int[] b = new int[n/2];
24          for (int i = 0; i <= last; ++i)
25              b[i] = arr[i];
26          for (int i = first; i < n; ++i)
27              b[i-n/2] = arr[i];
28          first = first-n/2;
29          arr = b;
30      }
31      return result;
32  }
```

Beobachtung

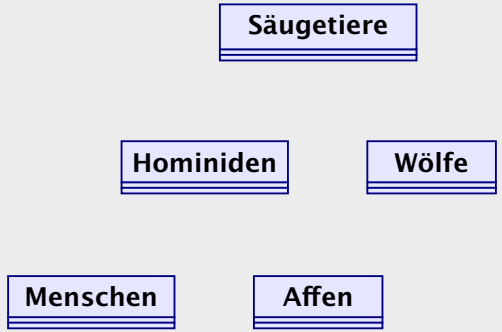
Oft werden mehrere Klassen von Objekten benötigt, die zwar ähnlich, aber doch verschieden sind.



12 Vererbung

Beobachtung

Oft werden mehrere Klassen von Objekten benötigt, die zwar ähnlich, aber doch verschieden sind.



12 Vererbung

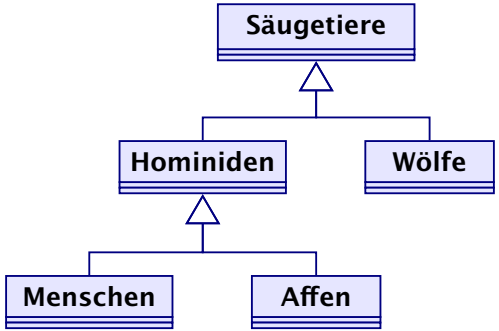
Idee:

- Finde Gemeinsamkeiten heraus!
- Organisiere in einer Hierarchie!
- Implementiere zuerst was allen gemeinsam ist!
- Implementiere dann nur noch den Unterschied!

$\Longrightarrow$ inkrementelles Programmieren

$\Longrightarrow$ Software Reuse

12 Vererbung



12 Vererbung

Idee:

- Finde Gemeinsamkeiten heraus!
- Organisiere in einer Hierarchie!
- Implementiere zuerst was allen gemeinsam ist!
- Implementiere dann nur noch den Unterschied!

$\Longrightarrow$ inkrementelles Programmieren

$\Longrightarrow$ Software Reuse

12 Vererbung

12 Vererbung

Prinzip

- Die Unterklasse verfügt über all Members der Oberklasse und eventuell noch über weitere.
- Das Übernehmen von Members der Oberklasse in die Unterklasse nennt man Vererbung (oder inheritance).

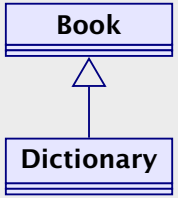
Beispiel

12 Vererbung

Prinzip

- Die Unterklasse verfügt über all Members der Oberklasse und eventuell noch über weitere.
- Das Übernehmen von Members der Oberklasse in die Unterklasse nennt man Vererbung (oder inheritance).

Beispiel



Implementierung

```
1  public class Book {
2      protected int pages;
3      public Book() {
4          pages = 150;
5      }
6      public void page_message() {
7          System.out.println("Number of pages: "+pages);
8      }
9  } // end of class Book
10 // continued...
```

Implementierung

```
1  public class Dictionary extends Book {
2      private int defs;
3      public Dictionary(int x) {
4          pages = 2*pages;
5          defs = x;
6      }
7      public void defs_message() {
8          System.out.println("Number of defs: "+defs);
9          System.out.println("Defs per page: "+defs/pages);
10     }
11 } // end of class Dictionary
```


Implementierung

```
1  public class Book {
2      protected int pages;
3      public Book() {
4          pages = 150;
5      }
6      public void page_message() {
7          System.out.println("Number of pages: "+pages);
8      }
9  } // end of class Book
10 // continued...
```

Implementierung

```
1  public class Dictionary extends Book {
2      private int defs;
3      public Dictionary(int x) {
4          pages = 2*pages;
5          defs = x;
6      }
7      public void defs_message() {
8          System.out.println("Number of defs: "+defs);
9          System.out.println("Defs per page: "+defs/pages);
10     }
11 } // end of class Dictionary
```


Erläuterungen

- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der Unterklasse nicht direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse A aufgerufen wird, wird implizit zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

Erläuterungen

- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der Unterklasse nicht direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse A aufgerufen wird, wird implizit zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

Beispiel

```
Dictionary webster = new Dictionary(12400);
```
liefert

Erläuterungen

- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der Unterklasse nicht direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse A aufgerufen wird, wird implizit zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

Beispiel

```
Dictionary webster = new Dictionary(12400);
```
liefert

Erläuterungen

- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der Unterklasse nicht direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse A aufgerufen wird, wird implizit zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

Beispiel

```
Dictionary webster = new Dictionary(12400);
```

liefert

Beispiel

```
Dictionary webster = new Dictionary(12400);
```
liefert

Erläuterungen

- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der Unterklasse nicht direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse A aufgerufen wird, wird implizit zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

Beispiel

```
Dictionary webster = new Dictionary(12400);
```
liefert

Erläuterungen

- `class A extends B { ... }` deklariert die Klasse A als Unterklasse der Klasse B.
- Alle Members von B stehen damit automatisch auch der Klasse A zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse sichtbar.
- Als `private` deklarierte Members können dagegen in der Unterklasse nicht direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse A aufgerufen wird, wird implizit zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

Methodenaufruf

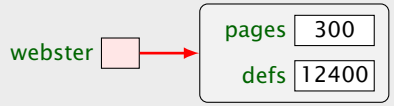
```
1 public class Words {
2     public static void main(String[] args) {
3         Dictionary webster = new Dictionary(12400);
4         webster.page_message();
5         webster.defs_message();
6     } // end of main
7 } // end of class Words
```

- Das neue Objekt `webster` enthält die Attribute `pages` und `defs`, sowie die Objekt-Methoden `page_message()` und `defs_message()`.
- Kommen in der Unterklasse nur weitere Members hinzu, spricht man von einer is_a-Beziehung. (Oft müssen aber Objekt-Methoden der Oberklasse in der Unterklasse umdefiniert werden.)

Beispiel

```
Dictionary webster = new Dictionary(12400);
```

liefert



Methodenaufruf

Die Programmausführung liefert:

```
Number of pages: 300
Number of defs: 12400
Defs per page: 41
```


Methodenaufruf

```
1 public class Words {
2     public static void main(String[] args) {
3         Dictionary webster = new Dictionary(12400);
4         webster.page_message();
5         webster.defs_message();
6     } // end of main
7 } // end of class Words
```

- Das neue Objekt webster enthält die Attribute pages und defs, sowie die Objekt-Methoden page_message() und defs_message().
- Kommen in der Unterklasse nur weitere Members hinzu, spricht man von einer is_a-Beziehung. (Oft müssen aber Objekt-Methoden der Oberklasse in der Unterklasse umdefiniert werden.)

12.1 Das Schlüsselwort super

- Manchmal ist es erforderlich, in der Unterklasse explizit die Konstruktoren oder Objekt-Methoden der Oberklasse aufzurufen. Das ist der Fall, wenn
  - Konstruktoren der Oberklasse aufgerufen werden sollen, die Parameter besitzen;
  - Objekt-Methoden oder Attribute der Oberklasse und Unterklasse gleiche Namen haben.
- Zur Unterscheidung der aktuellen Klasse von der Oberklasse dient das Schlüsselwort `super`.

Methodenaufruf

Die Programmausführung liefert:

```
Number of pages: 300
Number of defs: 12400
Defs per page: 41
```

Beispiel

```
1  public class Book {
2      protected int pages;
3      public Book(int x) {
4          pages = x;
5      }
6      public void message() {
7          System.out.println("Number of pages: "+pages);
8      }
9  } // end of class Book
10 //continued...
```


12.1 Das Schlüsselwort super

- Manchmal ist es erforderlich, in der Unterklasse explizit die Konstruktoren oder Objekt-Methoden der Oberklasse aufzurufen. Das ist der Fall, wenn
  - Konstruktoren der Oberklasse aufgerufen werden sollen, die Parameter besitzen;
  - Objekt-Methoden oder Attribute der Oberklasse und Unterklasse gleiche Namen haben.
- Zur Unterscheidung der aktuellen Klasse von der Oberklasse dient das Schlüsselwort `super`.

Beispiel

```
1  public class Book {
2      protected int pages;
3      public Book(int x) {
4          pages = x;
5      }
6      public void message() {
7          System.out.println("Number of pages: "+pages);
8      }
9  } // end of class Book
10 //continued...
```


Beispiel

```
11 public class Dictionary extends Book {
12     private int defs;
13     public Dictionary(int p, int d) {
14         super(p);
15         defs = d;
16     }
17     public void message() {
18         super.message();
19         System.out.println("Number of defs: "+defs);
20         System.out.println("Defs per page: "+defs/pages);
21     }
22 } // end of class Dictionary
```

Erläuterungen

„super“ als Konstruktoraufruf

- `super(...);` ruft den entsprechenden Konstruktor der Oberklasse auf.
- Analog gestattet `this(...);` den entsprechenden Konstruktor der eigenen Klasse aufzurufen.
- Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.

Beispiel

```
11 public class Dictionary extends Book {
12     private int defs;
13     public Dictionary(int p, int d) {
14         super(p);
15         defs = d;
16     }
17     public void message() {
18         super.message();
19         System.out.println("Number of defs: "+defs);
20         System.out.println("Defs per page: "+defs/pages);
21     }
22 } // end of class Dictionary
```

Erläuterungen

„super.“ zum Zugriff auf members der Oberklasse

Deklariert eine Klasse A einen Member `memb` gleichen Namens wie in einer Oberklasse, so ist nur noch der Member `memb` aus A sichtbar.

- Methoden mit unterschiedlichen Argument-Typen werden als verschieden angesehen.
- `super.memb` greift für das aktuelle Objekt `this` auf Attribute oder Objekt-Methoden `memb` der Oberklasse zu.
- Eine andere Verwendung von `super.` ist nicht gestattet.

Erläuterungen

„super“ als Konstruktoraufruf

- `super(...);` ruft den entsprechenden Konstruktor der Oberklasse auf.
- Analog gestattet `this(...);` den entsprechenden Konstruktor der eigenen Klasse aufzurufen.
- Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.

Verschattung von Variablen

Falls `memb` eine Methode ist:

- Wenn `memb` eine Methode mit den gleichen Argumenttypen (in der gleichen Reihenfolge), und dem gleichen Rückgabetypen ist, dann ist zunächst nur `memb` aus `A` sichtbar (Überschreiben).
- Wenn `memb` eine Methode mit unterschiedlichen Argumenttypen ist, dann sind sowohl `memb` aus `A` als auch die Methode der Oberklasse sichtbar (Überladen).
- Wenn die Argumenttypen übereinstimmen, aber der Rückgabetyp nicht, dann erhält man einen Compilerfehler.

Erläuterungen

„`super.`“ zum Zugriff auf members der Oberklasse

Deklariert eine Klasse A einen Member `memb` gleichen Namens wie in einer Oberklasse, so ist nur noch der Member `memb` aus A sichtbar.

- Methoden mit unterschiedlichen Argument-Typen werden als verschieden angesehen.
- `super.memb` greift für das aktuelle Objekt `this` auf Attribute oder Objekt-Methoden `memb` der Oberklasse zu.
- Eine andere Verwendung von `super.` ist nicht gestattet.

Verschattung von Variablen

Falls `memb` eine Variable ist:

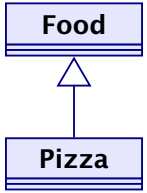
- Direkt (d.h. ohne `super.`) ist nur `memb` aus `A` sichtbar. `memb` kann einen anderen Typ als in der Oberklasse haben.

Verschattung von Variablen

Falls `memb` eine Methode ist:

- Wenn `memb` eine Methode mit den gleichen Argumenttypen (in der gleichen Reihenfolge), und dem gleichen Rückgabetypen ist, dann ist zunächst nur `memb` aus `A` sichtbar (Überschreiben).
- Wenn `memb` eine Methode mit unterschiedlichen Argumenttypen ist, dann sind sowohl `memb` aus `A` als auch die Methode der Oberklasse sichtbar (Überladen).
- Wenn die Argumenttypen übereinstimmen, aber der Rückgabetyp nicht, dann erhält man einen Compilerfehler.

12.2 Private Variablen und Methoden



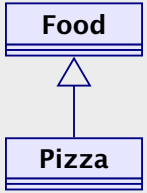
Das Programm `Eating` soll die Anzahl der Kalorien pro Mahlzeit ausgeben.

Verschattung von Variablen

Falls `memb` eine Variable ist:

- Direkt (d.h. ohne `super.`) ist nur `memb` aus A sichtbar. `memb` kann einen anderen Typ als in der Oberklasse haben.

12.2 Private Variablen und Methoden



Das Programm Eating soll die Anzahl der Kalorien pro Mahlzeit ausgeben.

Implementierung

```
1 public class Eating {
2     public static void main (String[] args) {
3         Pizza special = new Pizza(275);
4         System.out.print("Calories per serving: " +
              special.caloriesPerServing());
5     } // end of main
6 } // end of class Eating
```

Implementierung

```
7  public class Food {
8      private int CALORIES_PER_GRAM = 9;
9      private int fat, servings;
10     public Food (int numFatGrams, int numServings) {
11         fat = numFatGrams;
12         servings = numServings;
13     }
14     private int calories() {
15         return fat * CALORIES_PER_GRAM;
16     }
17     public int caloriesPerServing() {
18         return calories() / servings;
19     }
20 } // end of class Food
```


Implementierung

```
1  public class Eating {
2      public static void main (String[] args) {
3          Pizza special = new Pizza(275);
4          System.out.print("Calories per serving: " +
               special.caloriesPerServing());
5      } // end of main
6  } // end of class Eating
```

Implementierung + Erläuterungen

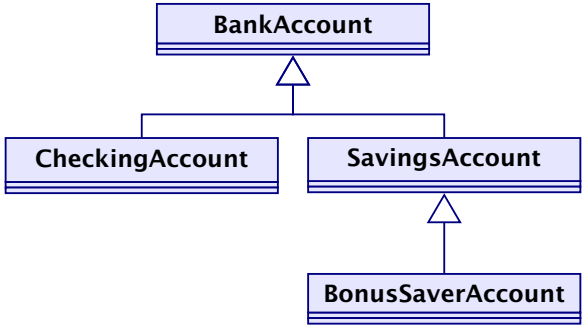
```
21 public class Pizza extends Food {
22     public Pizza (int amountFat) {
23         super(amountFat,8);
24     }
25 } // end of class Pizza
```

- Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` — nicht alle direkt zugänglich.
- Die Attribute und die Objekt-Methode `calories()` der Klasse `Food` sind privat, und damit für Objekte der Klasse `Pizza` verborgen.
- Trotzdem können sie von der `public` Objekt-Methode `caloriesPerServing` benutzt werden.

Ausgabe des Programms: `Calories per serving: 309`

Implementierung

```
 7 public class Food {
 8     private int CALORIES_PER_GRAM = 9;
 9     private int fat, servings;
10     public Food (int numFatGrams, int numServings) {
11         fat = numFatGrams;
12         servings = numServings;
13     }
14     private int calories() {
15         return fat * CALORIES_PER_GRAM;
16     }
17     public int caloriesPerServing() {
18         return calories() / servings;
19     }
20 } // end of class Food
```



Implementierung + Erläuterungen

```
21 public class Pizza extends Food {
22     public Pizza (int amountFat) {
23         super(amountFat,8);
24     }
25 } // end of class Pizza
```

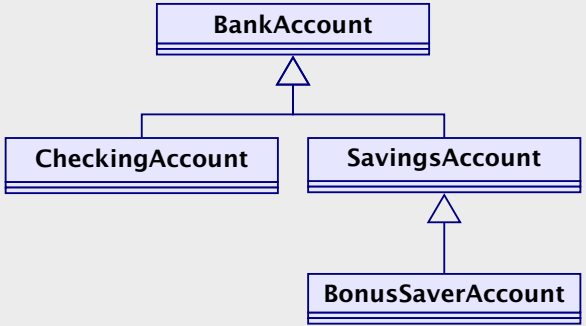
- Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` — nicht alle direkt zugänglich.
- Die Attribute und die Objekt-Methode `calories()` der Klasse `Food` sind privat, und damit für Objekte der Klasse `Pizza` verborgen.
- Trotzdem können sie von der `public` Objekt-Methode `caloriesPerServing` benutzt werden.

Ausgabe des Programms: `Calories per serving: 309`

Aufgabe

- Implementierung von einander abgeleiteter Formen von Bankkonten.
- Jedes Konto kann eingerichtet werden, erlaubt Einzahlungen und Auszahlungen.
- Verschiedene Konten verhalten sich unterschiedlich in Bezug auf Zinsen und Kosten von Kontobewegungen.

12.3 Überschreiben von Methoden



Einige Konten

```
1  public class Bank {
2      public static void main(String[] args) {
3          SavingsAccount savings =
4              new SavingsAccount(4321, 5028.45, 0.02);
5          BonusSaverAccount bigSavings =
6              new BonusSaverAccount (6543, 1475.85, 0.02);
7          CheckingAccount checking =
8              new CheckingAccount (9876,269.93, savings);
9          savings.deposit(148.04);     System.out.println();
10         bigSavings.deposit(41.52);   System.out.println();
11         savings.withdraw(725.55);    System.out.println();
12         bigSavings.withdraw(120.38); System.out.println();
13         checking.withdraw(320.18);   System.out.println();
14     } // end of main
15 } // end of class Bank
```

"Bank.java"

Aufgabe

- Implementierung von einander abgeleiteter Formen von Bankkonten.
- Jedes Konto kann eingerichtet werden, erlaubt Einzahlungen und Auszahlungen.
- Verschiedene Konten verhalten sich unterschiedlich in Bezug auf Zinsen und Kosten von Kontobewegungen.

Implementierung

```
1  public class BankAccount {
2      // Attribute aller Konten-Klassen:
3      protected int account;
4      protected double balance;
5      // Konstruktor:
6      public BankAccount(int id, double initial) {
7          account = id; balance = initial;
8      }
9      // Objekt-Methoden:
10     public void deposit(double amount) {
11         balance = balance + amount;
12         System.out.println(
13             "Deposit into account "    + account +"\n"
14                     + "Amount:\t\t"    + amount  +"\n"
15                     + "New balance:\t" + balance);
16     }
```

"BankAccount.java"

Einige Konten

```
1  public class Bank {
2      public static void main(String[] args) {
3          SavingsAccount savings =
4              new SavingsAccount(4321, 5028.45, 0.02);
5          BonusSaverAccount bigSavings =
6              new BonusSaverAccount (6543, 1475.85, 0.02);
7          CheckingAccount checking =
8              new CheckingAccount (9876,269.93, savings);
9          savings.deposit(148.04);     System.out.println();
10         bigSavings.deposit(41.52);   System.out.println();
11         savings.withdraw(725.55);    System.out.println();
12         bigSavings.withdraw(120.38); System.out.println();
13         checking.withdraw(320.18);   System.out.println();
14     } // end of main
15 } // end of class Bank
```

"Bank.java"

Erläuterungen

- Anlegen eines Kontos BankAccount speichert eine (hoffentlich neue) Kontonummer sowie eine Anfangseinlage.
- Die zugehörigen Attribute sind protected, d.h. können nur von Objekt-Methoden der Klasse bzw. ihrer Unterklassen modifiziert werden.
- die Objekt-Methode deposit legt Geld aufs Konto, d.h. modifiziert den Wert von balance und teilt die Kontobewegung mit.

Implementierung

```
1  public class BankAccount {
2      // Attribute aller Konten-Klassen:
3      protected int account;
4      protected double balance;
5      // Konstruktor:
6      public BankAccount(int id, double initial) {
7          account = id; balance = initial;
8      }
9      // Objekt-Methoden:
10     public void deposit(double amount) {
11         balance = balance + amount;
12         System.out.println(
13             "Deposit into account "     + account +"\n"
14                     + "Amount:\t\t"     + amount  +"\n"
15                     + "New balance:\t" + balance);
16     }
```

"BankAccount.java"

Implementierung

```
17      public boolean withdraw(double amount) {
18          System.out.println(
19              "Withdrawal from account "+ account +"\n"
20                              + "Amount:\t\t" + amount);
21          if (amount > balance) {
22              System.out.println(
23                  "Sorry, insufficient funds...");
24              return false;
25          }
26          balance = balance - amount;
27          System.out.println(
28              "New balance:\t"+ balance);
29          return true;
30      }
31  } // end of class BankAccount
```

"BankAccount.java"

Erläuterungen

- Anlegen eines Kontos `BankAccount` speichert eine (hoffentlich neue) Kontonummer sowie eine Anfangseinlage.
- Die zugehörigen Attribute sind `protected`, d.h. können nur von Objekt-Methoden der Klasse bzw. ihrer Unterklassen modifiziert werden.
- die Objekt-Methode `deposit` legt Geld aufs Konto, d.h. modifiziert den Wert von `balance` und teilt die Kontobewegung mit.

Erläuterungen

- Die Objekt-Methode `withdraw()` nimmt eine Auszahlung vor.
- Falls die Auszahlung scheitert, wird eine Mitteilung gemacht.
- Ob die Auszahlung erfolgreich war, teilt der Rückgabewert mit.
- Ein `CheckingAccount` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.

Implementierung

```
17     public boolean withdraw(double amount) {
18         System.out.println(
19             "Withdrawal from account "+ account +"\n"
20                                 + "Amount:\t\t" + amount);
21         if (amount > balance) {
22             System.out.println(
23                 "Sorry, insufficient funds...");
24             return false;
25         }
26         balance = balance - amount;
27         System.out.println(
28             "New balance:\t"+ balance);
29         return true;
30     }
31 } // end of class BankAccount
```

"BankAccount.java"

Erläuterungen

- Die Objekt-Methode `withdraw()` nimmt eine Auszahlung vor.
- Falls die Auszahlung scheitert, wird eine Mitteilung gemacht.
- Ob die Auszahlung erfolgreich war, teilt der Rückgabewert mit.
- Ein `CheckingAccount` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.

Ein Girokonto

```
1 public class CheckingAccount extends BankAccount {
2     private SavingsAccount overdraft;
3     // Konstruktor:
4     public CheckingAccount(int id, double initial,
          SavingsAccount savings) {
5         super(id, initial);
6         overdraft = savings;
7     }
```

"CheckingAccount.java"

Modifiziertes `withdraw()`

```
8     // modifiziertes withdraw():
9     public boolean withdraw(double amount) {
10        if (!super.withdraw(amount)) {
11            System.out.println("Using overdraft...");
12            if (!overdraft.withdraw(amount-balance)) {
13                System.out.println(
14                    "Overdraft source insufficient.");
15                return false;
16            } else {
17                balance = 0;
18                System.out.println(
19                    "New balance on account "
20                    + account + ": 0");
21            }
22        }
23        return true;
24    }
25 } // end of class CheckingAccount
```

"CheckingAccount.java"

Ein Girokonto

```
1  public class CheckingAccount extends BankAccount {
2      private SavingsAccount overdraft;
3      // Konstruktor:
4      public CheckingAccount(int id, double initial,
           SavingsAccount savings) {
5          super(id, initial);
6          overdraft = savings;
7      }
```

"CheckingAccount.java"

Erläuterungen

- Die Objekt-Methode `withdraw` wird neu definiert, die Objekt-Methode `deposit` wird übernommen.
- Der Normalfall des Abhebens erfolgt (als Seiteneffekt) beim Testen der ersten `if`-Bedingung.
- Dazu wird die `withdraw`-Methode der Oberklasse aufgerufen.
- Scheitert das Abheben mangels Geldes, wird der Fehlbetrag vom Rücklagen-Konto abgehoben.
- Scheitert auch das, erfolgt keine Konto-Bewegung, dafür eine Fehlermeldung.
- Andernfalls sinkt der aktuelle Kontostand auf 0 und die Rücklage wird verringert.

Modifiziertes `withdraw()`

```
8     // modifiziertes withdraw():
9     public boolean withdraw(double amount) {
10        if (!super.withdraw(amount)) {
11            System.out.println("Using overdraft...");
12            if (!overdraft.withdraw(amount-balance)) {
13                System.out.println(
14                    "Overdraft source insufficient.");
15                return false;
16            } else {
17                balance = 0;
18                System.out.println(
19                    "New balance on account "
20                    + account + ": 0");
21            }
22        }
23        return true;
24    }
25 } // end of class CheckingAccount
```

"CheckingAccount.java"

Ein Sparbuch

```
1  public class SavingsAccount extends BankAccount {
2      protected double interestRate;
3      // Konstruktor:
4      public SavingsAccount(int id,double init,double rate){
5          super(id, init);
6          interestRate = rate;
7      }
8      // zusaetzliche Objekt-Methode:
9      public void addInterest() {
10         balance = balance * (1 + interestRate);
11         System.out.println(
12             "Interest added to account: "+ account
13             + "\nNew balance:\t" + balance);
14     }
15 } // end of class SavingsAccount
```

"SavingsAccount.java"

Erläuterungen

- Die Objekt-Methode `withdraw` wird neu definiert, die Objekt-Methode `deposit` wird übernommen.
- Der Normalfall des Abhebens erfolgt (als Seiteneffekt) beim Testen der ersten `if`-Bedingung.
- Dazu wird die `withdraw`-Methode der Oberklasse aufgerufen.
- Scheitert das Abheben mangels Geldes, wird der Fehlbetrag vom Rücklagen-Konto abgehoben.
- Scheitert auch das, erfolgt keine Konto-Bewegung, dafür eine Fehlermeldung.
- Andernfalls sinkt der aktuelle Kontostand auf 0 und die Rücklage wird verringert.

Ein Sparbuch

```
1  public class SavingsAccount extends BankAccount {
2      protected double interestRate;
3      // Konstruktor:
4      public SavingsAccount(int id,double init,double rate){
5          super(id, init);
6          interestRate = rate;
7      }
8      // zusaetzliche Objekt-Methode:
9      public void addInterest() {
10         balance = balance * (1 + interestRate);
11         System.out.println(
12             "Interest added to account: "+ account
13             + "\nNew balance:\t" + balance);
14     }
15 } // end of class SavingsAccount
```

"SavingsAccount.java"

Erläuterungen

- Die Klasse SavingsAccount erweitert die Klasse BankAccount um das zusätzliche Attribut double interestRate (Zinssatz) und eine Objekt-Methode, die die Zinsen gutschreibt.
- Alle sonstigen Attribute und Objekt-Methoden werden von der Oberklasse geerbt.
- Die Klasse BonusSaverAccount erhöht zusätzlich den Zinssatz, führt aber Strafkosten fürs Abheben ein.

Ein Bonus-Sparbuch

```
1  public class BonusSaverAccount extends SavingsAccount {
2      private int penalty;
3      private double bonus;
4      // Konstruktor:
5      public BonusSaverAccount(int id, double init,
6                                       double rate) {
7          super(id, init, rate);
8          penalty = 25;
9          bonus = 0.03;
10     }
11     // Modifizierung der Objekt-Methoden:
12     public boolean withdraw(double amount) {
13         boolean res;
14         if (res = super.withdraw(amount + penalty))
15             System.out.println(
16                 "Penalty incurred:\t"+ penalty);
17         return res;
18     }
```

"BonusSaverAccount.java"

Erläuterungen

- Die Klasse `SavingsAccount` erweitert die Klasse `BankAccount` um das zusätzliche Attribut `double interestRate` (Zinssatz) und eine Objekt-Methode, die die Zinsen gutschreibt.
- Alle sonstigen Attribute und Objekt-Methoden werden von der Oberklasse geerbt.
- Die Klasse `BonusSaverAccount` erhöht zusätzlich den Zinssatz, führt aber Strafkosten fürs Abheben ein.

Ein Bonus-Sparbuch

```
19     public void addInterest() {
20         balance = balance * (1 + interestRate + bonus);
21         System.out.println(
22                 "Interest added to account: " + account
23             + "\nNew balance:\t" + balance);
24     }
25 } // end of class BonusSaverAccount
```

"BonusSaverAccount.java"

Ein Bonus-Sparbuch

```
1  public class BonusSaverAccount extends SavingsAccount {
2      private int penalty;
3      private double bonus;
4      // Konstruktor:
5      public BonusSaverAccount(int id, double init,
6                                           double rate) {
7          super(id, init, rate);
8          penalty = 25;
9          bonus = 0.03;
10     }
11     // Modifizierung der Objekt-Methoden:
12     public boolean withdraw(double amount) {
13         boolean res;
14         if (res = super.withdraw(amount + penalty))
15             System.out.println(
16                 "Penalty incurred:\t"+ penalty);
17         return res;
18     }
```

"BonusSaverAccount.java"

Programmausgabe

```
Deposit into account 4321
Amount: 148.04
New balance: 5176.49

Deposit into account 6543
Amount: 41.52
New balance: 1517.37

Withdrawal from account 4321
Amount: 725.55
New balance: 4450.94

Withdrawal from account 6543
Amount: 145.38
New balance: 1371.989999999998
Penalty incurred: 25

Withdrawal from account 9876
Amount: 320.18
Sorry, insufficient funds...
Using overdraft...
Withdrawal from account 4321
Amount: 50.25
New balance: 4400.69
New balance on account 9876: 0
```

Ein Bonus-Sparbuch

```
19      public void addInterest() {
20          balance = balance * (1 + interestRate + bonus);
21          System.out.println(
22                  "Interest added to account: " + account
23              + "\nNew balance:\t" + balance);
24      }
25  } // end of class BonusSaverAccount
```

"BonusSaverAccount.java"

13 Polymorphie

Problem:

- Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur grade nochmal definieren.

13 Polymorphie

Problem:

- Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur grade nochmal definieren.

13.1 Unterklassen-Polymorphie

Idee:
Überall wo ein Objekt vom Typ `ClassA` verwendet wird, können wir auch ein Objekt einer Unterklasse von `ClassA` nutzen.

- Zuweisungen:

```
ClassA a;
ClassB b = new ClassB();
a = b; // weise Objekt einer Unterklasse zu
```

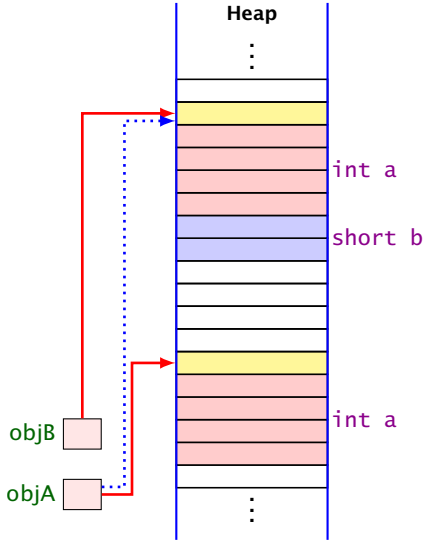
- Methodenaufrufe:

```
void meth(ClassA a) {};
ClassB b = new ClassB();
void bla() {
    meth(b); // rufe meth mit Objekt von
             // Unterklasse auf
}
```

Was passiert hier eigentlich?

```
public ClassA {
    int a;
}
public ClassB extends classA {
    short b;
}
public class Test {
    public static void main() {
        ClassA objA = new ClassA();
        ClassB objB = new ClassB();
        objA = objB;
    }
}
```

Ein Objekt vom Typ `classB` ist auch ein Objekt vom Typ `classA`.



13.1 Unterklassen-Polymorphie

Idee:
Überall wo ein Objekt vom Typ `ClassA` verwendet wird, können wir auch ein Objekt einer Unterklasse von `ClassA` nutzen.

- Zuweisungen:

```
ClassA a;
ClassB b = new ClassB();
a = b; // weise Objekt einer Unterklasse zu
```

- Methodenaufrufe:

```
void meth(ClassA a) {};
ClassB b = new ClassB();
void bla() {
    meth(b); // rufe meth mit Objekt von
             // Unterklasse auf
}
```

Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10 € gutschreiben, um die Wirtschaft anzukurbeln:

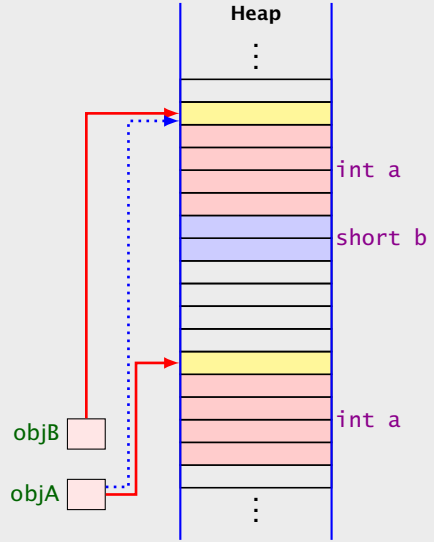
```
void boostEconomy(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].deposit(10);
    }
}
```

- Die Methode bekommt ein Array mit allen Konten übergeben.
- Die einzelnen Elemente des Arrays können BankAcccount, CheckingAccount, SavingsAccount, oder BonusSaverAccount sein.
- Es wird jeweils die Methode deposit aufgerufen, die in der Klasse BankAccount implementiert ist.

Was passiert hier eigentlich?

```
public ClassA {
    int a;
}
public ClassB extends classA {
    short b;
}
public class Test {
    public static void main() {
        ClassA objA = new ClassA();
        ClassB objB = new ClassB();
        objA = objB;
    }
}
```

Ein Objekt vom Typ `classB` ist auch ein Objekt vom Typ `classA`.



Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10 € gutschreiben, um die Wirtschaft anzukurbeln:

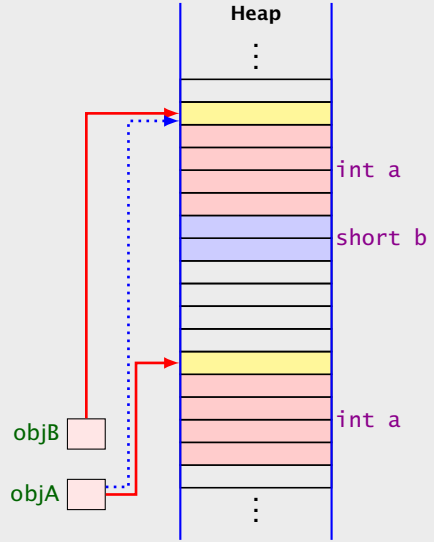
```
void boostEconomy(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].deposit(10);
    }
}
```

- Die Methode bekommt ein Array mit allen Konten übergeben.
- Die einzelnen Elemente des Arrays können BankAcccount, CheckingAccount, SavingsAccount, oder BonusSaverAccount sein.
- Es wird jeweils die Methode deposit aufgerufen, die in der Klasse BankAccount implementiert ist.

Was passiert hier eigentlich?

```
public ClassA {
    int a;
}
public ClassB extends classA {
    short b;
}
public class Test {
    public static void main() {
        ClassA objA = new ClassA();
        ClassB objB = new ClassB();
        objA = objB;
    }
}
```

Ein Objekt vom Typ classB ist auch ein Objekt vom Typ classA.



Realistisches Beispiel

Die Mafia ist durch ein Hack in den Besitz einer großen Menge von Bankdaten gekommen. Diese gilt es auszunutzen:

```
void exploitHack(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].withdraw(10);
    }
}
```

- Hier wird die (spezielle) withdraw-Methode des jeweiligen Account-Typs aufgerufen.
- Die kann der Compiler aber nicht kennen!!!
- Dynamische Methodenbindung!!!

Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10 € gutschreiben, um die Wirtschaft anzukurbeln:

```
void boostEconomy(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].deposit(10);
    }
}
```

- Die Methode bekommt ein Array mit allen Konten übergeben.
- Die einzelnen Elemente des Arrays können `BankAcccount`, `CheckingAccount`, `SavingsAccount`, oder `BonusSaverAccount` sein.
- Es wird jeweils die Methode `deposit` aufgerufen, die in der Klasse `BankAccount` implementiert ist.

Realistisches Beispiel

Die Mafia ist durch ein Hack in den Besitz einer großen Menge von Bankdaten gekommen. Diese gilt es auszunutzen:

```
void exploitHack(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].withdraw(10);
    }
}
```

- Hier wird die (spezielle) `withdraw`-Methode des jeweiligen Account-Typs aufgerufen.
- Die kann der Compiler aber nicht kennen!!!
- Dynamische Methodenbindung!!!

Unrealistisches Beispiel

Der Finanzminister möchte jedem Konto 10 € gutschreiben, um die Wirtschaft anzukurbeln:

```
void boostEconomy(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].deposit(10);
    }
}
```

- Die Methode bekommt ein Array mit allen Konten übergeben.
- Die einzelnen Elemente des Arrays können `BankAcccount`, `CheckingAccount`, `SavingsAccount`, oder `BonusSaverAccount` sein.
- Es wird jeweils die Methode `deposit` aufgerufen, die in der Klasse `BankAccount` implementiert ist.

Statischer vs. dynamischer Typ

Der statische Typ eines Ausdrucks ist, der Typ, der sich gemäß den Regeln zu Auswertung von Ausdrücken ergibt.

Der dynamische Typ eines Referenzausdrucks e ist der Typ des wirklichen Objekts auf dass e zur Laufzeit zeigt.

Beispiel:

```
SavingsAccount s = new SavingsAccount(89,10,0.2);
BankAccount b = s;

s //statischer Typ SavingsAccount
b //statischer Typ BankAccount

s //dynamischer Typ SavingsAccount
b //dynamischer Typ SavingsAccount
```

Realistisches Beispiel

Die Mafia ist durch ein Hack in den Besitz einer großen Menge von Bankdaten gekommen. Diese gilt es auszunutzen:

```
void exploitHack(BankAccount[] arr) {
    for (int i=0; i<arr.length, i++) {
        arr[i].withdraw(10);
    }
}
```

- Hier wird die (spezielle) withdraw-Methode des jeweiligen Account-Typs aufgerufen.
- Die kann der Compiler aber nicht kennen!!!
- Dynamische Methodenbindung!!!

Ermittlung der aufgerufenen Methode

Betrachte einen Aufruf $e_0.\mathrm{f}(e_1,\ldots,e_k)$.

1. Bestimme die statischen Typen $T_0,\ldots,T_k$ der Ausdrücke $e_0,\ldots,e_k$.
2. Suche in einer Oberklasse von T_0 nach einer Methode mit Namen f, deren Liste von Argumenttypen bestmöglich zu der Liste $T_1,\ldots,T_k$ passt.
   Sei S Signatur dieser rein statisch gefundenen Methode f.
3. Der dynamische Typ D des Objekts, zu dem sich e_0 auswertet, gehört zu einer Unterklasse von T_0.
4. Es wird die Methode f aufgerufen, die Signatur S hat, und die in der nächsten Oberklasse von D implementiert wird.

Statischer vs. dynamischer Typ

Der statische Typ eines Ausdrucks ist, der Typ, der sich gemäß den Regeln zu Auswertung von Ausdrücken ergibt.

Der dynamische Typ eines Referenzausdrucks e ist der Typ des wirklichen Objekts auf dass e zur Laufzeit zeigt.

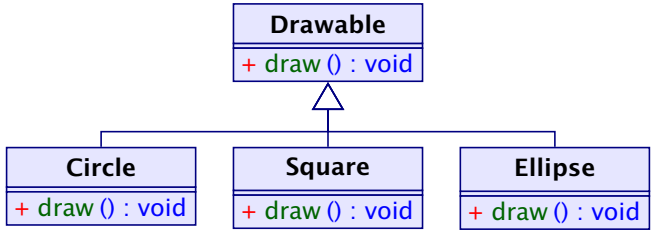
Beispiel:

```
SavingsAccount s = new SavingsAccount(89,10,0.2);
BankAccount b = s;

s //statischer Typ SavingsAccount
b //statischer Typ BankAccount

s //dynamischer Typ SavingsAccount
b //dynamischer Typ SavingsAccount
```

Weiteres Beispiel




```
1  public class Figure {
2      Drawable[] arr; // contains basic shapes of figure
3      Figure(/* some parameters */) {
4          /* constructor initializes arr */
5      }
6      void draw() {
7          for (int i=0; i<arr.length; ++i) {
8              a[i].draw();
9          }
10 }   }
```

Ermittlung der aufgerufenen Methode

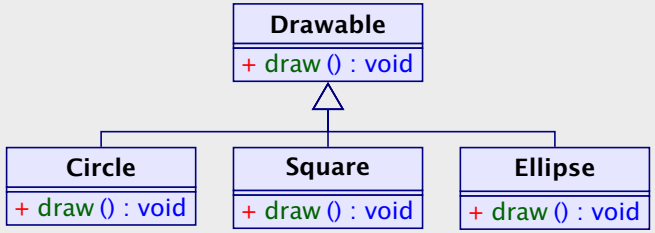
Betrachte einen Aufruf $e_0.\mathrm{f}(e_1,\ldots,e_k)$.

1. Bestimme die statischen Typen $T_0,\ldots,T_k$ der Ausdrücke $e_0,\ldots,e_k$.
2. Suche in einer Oberklasse von T_0 nach einer Methode mit Namen f, deren Liste von Argumenttypen bestmöglich zu der Liste $T_1,\ldots,T_k$ passt.
   Sei S Signatur dieser rein statisch gefundenen Methode f.
3. Der dynamische Typ D des Objekts, zu dem sich e_0 auswertet, gehört zu einer Unterklasse von T_0.
4. Es wird die Methode f aufgerufen, die Signatur S hat, und die in der nächsten Oberklasse von D implementiert wird.

Die Klasse Object

- Die Klasse `Object` ist eine gemeinsame Oberklasse für alle Klassen.
- Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.

Weiteres Beispiel




```
1  public class Figure {
2      Drawable[] arr; // contains basic shapes of figure
3      Figure(/* some parameters */) {
4          /* constructor initializes arr */
5      }
6      void draw() {
7          for (int i=0; i<arr.length; ++i) {
8              a[i].draw();
9          }
10 }   }
```

Die Klasse Object

Einige nützliche Methoden der Klasse `Object`:

- `String toString()` liefert Darstellung als `String`;
- `boolean equals(Object obj)` testet auf Objekt-Identität oder Referenz-Gleichheit:

```
1 public boolean equals(Object obj) {
2     return this == obj;
3 }
```

- `int hashCode()` liefert Nummer für das Objekt.
- . . . viele weitere geheimnisvolle Methoden, die u.a. mit paralleler Programmausführung zu tun haben.

Achtung: `Object`-Methoden können aber in Unterklassen durch geeignetere Methoden überschrieben werden.

Die Klasse Object

- Die Klasse `Object` ist eine gemeinsame Oberklasse für alle Klassen.
- Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.

Beispiel

```
1  public class Poly {
2      public String toString() { return "Hello"; }
3  }
4  public class PolyTest {
5      public static String addWorld(Object x) {
6          return x.toString() + " World!";
7      }
8      public static void main(String[] args) {
9          Object x = new Poly();
10         System.out.println(addWorld(x));
11     }
12 }
```

liefert: "Hello World!"

Die Klasse Object

Einige nützliche Methoden der Klasse Object:

- String toString() liefert Darstellung als String;
- boolean equals(Object obj) testet auf Objekt-Identität oder Referenz-Gleichheit:

```
1 public boolean equals(Object obj) {
2     return this == obj;
3 }
```

- int hashCode() liefert Nummer für das Objekt.
- ... viele weitere geheimnisvolle Methoden, die u.a. mit paralleler Programmausführung zu tun haben.

Achtung: Object-Methoden können aber in Unterklassen durch geeignetere Methoden überschrieben werden.

Beispiel

```
1  public class Poly {
2      public String toString() { return "Hello"; }
3  }
4  public class PolyTest {
5      public static String addWorld(Object x) {
6          return x.toString() + " World!";
7      }
8      public static void main(String[] args) {
9          Object x = new Poly();
10         System.out.println(addWorld(x));
11     }
12 }
```

liefert: "Hello World!"

Erläuterungen

- Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- Die Klasse `Poly` ist eine Unterklasse von `Object`.
- Einer Variable der Klasse `ClassA` kann ein Objekt jeder Unterklasse von `ClassA` zugewiesen werden.
- Darum kann `x` das neue `Poly`-Objekt aufnehmen.

Erläuterungen

- Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- Die Klasse `Poly` ist eine Unterklasse von `Object`.
- Einer Variable der Klasse `ClassA` kann ein Objekt jeder Unterklasse von `ClassA` zugewiesen werden.
- Darum kann `x` das neue `Poly`-Objekt aufnehmen.

Beispiel

```
1 public class PolyB {
2       public String greeting() { return "Hello"; }
3 }
4 public class PolyTestB {
5       public static void main(String[] args) {
6           Object x = new PolyB();
7           System.out.println(x.greeting()+" World!");
8       }
9 }
```

liefert: Compilerfehler

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                               ^
1 error
```

Beispiel

```
1 public class PolyB {
2       public String greeting() { return "Hello"; }
3 }
4 public class PolyTestB {
5       public static void main(String[] args) {
6           Object x = new PolyB();
7           System.out.println(x.greeting()+" World!");
8       }
9 }
```

liefert: Compilerfehler

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                                 ^
1 error
```


Erklärung

- Die Variable x ist als Object deklariert.
- Der Compiler weiss nicht, ob der aktuelle Wert von x ein Objekt aus einer Unterklasse ist, in welcher die Objektmethode greeting() definiert ist.
- Darum lehnt er dieses Programm ab.

Erklärung

- Die Variable `x` ist als `Object` deklariert.
- Der Compiler weiss nicht, ob der aktuelle Wert von `x` ein Objekt aus einer Unterklasse ist, in welcher die Objektmethode `greeting()` definiert ist.
- Darum lehnt er dieses Programm ab.

Methodenaufruf

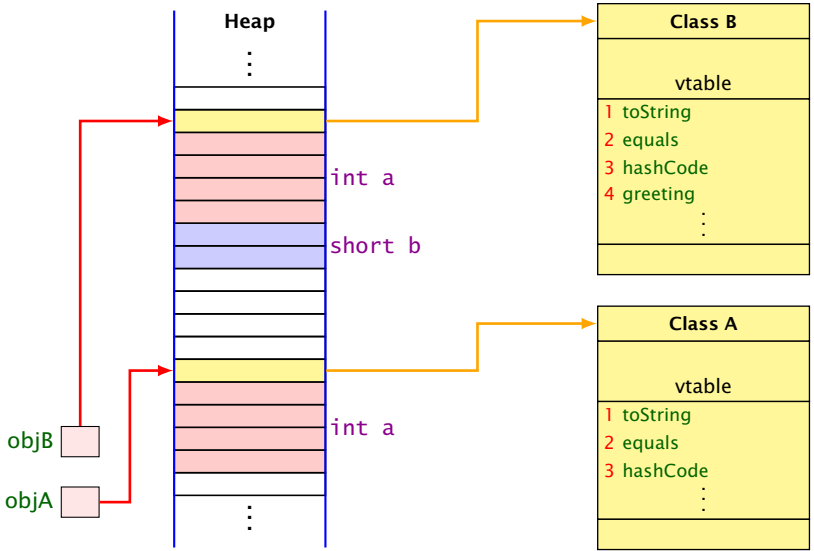
Der Aufruf einer statischen Methode:

1. Aktuelle Parameter und Rücksprungadresse auf den Stack legen.
2. Zum Code der Funktion springen.

Aufruf einer Objektmethode:

1. Aktuelle Parameter (auch `this`) und Rücksprungadresse auf den Stack legen.
2. **Problem:** Die aufgerufene Funktion ist zur Compilezeit noch nicht bekannt; existiert vielleicht nicht einmal.

Methodenaufruf



Methodenaufruf

Der Aufruf einer statischen Methode:

1. Aktuelle Parameter und Rücksprungadresse auf den Stack legen.
2. Zum Code der Funktion springen.

Aufruf einer Objektmethode:

1. Aktuelle Parameter (auch `this`) und Rücksprungadresse auf den Stack legen.
2. **Problem:** Die aufgerufene Funktion ist zur Compilezeit noch nicht bekannt; existiert vielleicht nicht einmal.

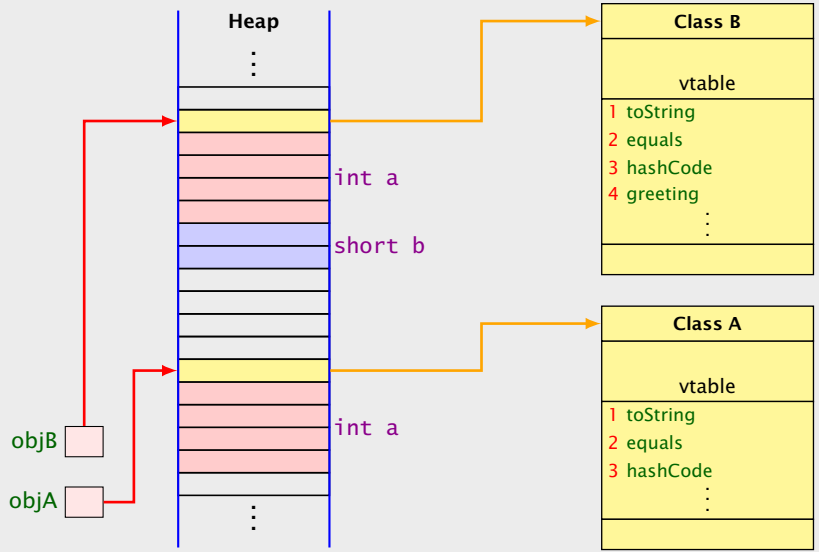
Methodenaufruf

- Jede Klasse hat eine Tabelle (`vtable`) mit Methoden, die zu dieser Klasse gehören. Darin wird die Adresse des zugehörigen Codes gespeichert.
- Ein Aufruf einer Objektmethode (z.B. `equals`) sucht in dieser Tabelle nach der Sprungadresse.
- Beim Überschreiben einer Methode in einer Unterklasse wird dieser Eintrag auf die Sprungadresse der neuen Funktion geändert.
- Dynamische Methodenbindung

Wichtig

Der Index der Funktionen innerhalb der (`vtable`) ist in jeder abgeleiteten Klasse gleich.

Methodenaufruf



Beispiel

```
1 public class PolyB {
2       public String greeting() { return "Hello"; }
3 }
4 public class PolyTestB {
5       public static void main(String[] args) {
6           Object x = new PolyB();
7           System.out.println(x.greeting()+" World!");
8       }
9 }
```

liefert: Compilerfehler

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                                 ^
1 error
```


Methodenaufruf

- Jede Klasse hat eine Tabelle (vtable) mit Methoden, die zu dieser Klasse gehören. Darin wird die Adresse des zugehörigen Codes gespeichert.
- Ein Aufruf einer Objektmethode (z.B. equals) sucht in dieser Tabelle nach der Sprungadresse.
- Beim Überschreiben einer Methode in einer Unterklasse wird dieser Eintrag auf die Sprungadresse der neuen Funktion geändert.
- Dynamische Methodenbindung

Wichtig

Der Index der Funktionen innerhalb der (vtable) ist in jeder abgeleiteten Klasse gleich.

Beispiel

```
1 public class PolyB {
2       public String greeting() { return "Hello"; }
3 }
4 public class PolyTestB {
5       public static void main(String[] args) {
6           Object x = new PolyB();
7           System.out.println(x.greeting()+" World!");
8       }
9 }
```

liefert: Compilerfehler

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                                 ^
1 error
```


Ausweg

Benutze einen expliziten cast in die entsprechende Unterklasse!

```
1 public class PolyC {
2     public String greeting() { return "Hello"; }
3 }
4 public class PolyTestC {
5     public void main(String[] args) {
6         Object x = new PolyC();
7         if (x instanceof PolyC)
8             System.out.print(((PolyC) x).greeting()+"
                  World!\n");
9         else
10            System.out.print("Sorry: no cast
                  possible!\n");
11    }
12 }
```

Ausweg

Benutze einen expliziten cast in die entsprechende Unterklasse!

```
1  public class PolyC {
2      public String greeting() { return "Hello"; }
3  }
4  public class PolyTestC {
5      public void main(String[] args) {
6          Object x = new PolyC();
7          if (x instanceof PolyC)
8              System.out.print(((PolyC) x).greeting()+"
                   World!\n");
9          else
10             System.out.print("Sorry: no cast
                   possible!\n");
11     }
12 }
```


Fazit

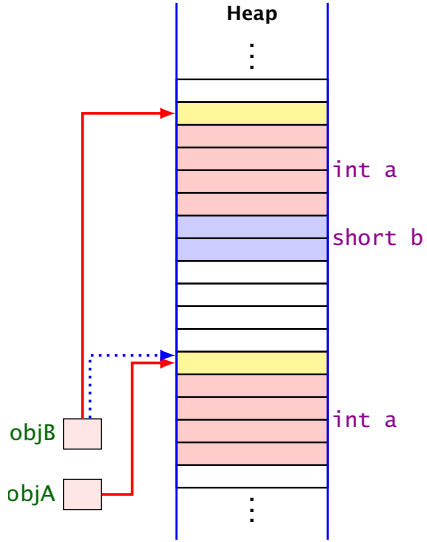
- Eine Variable x einer Klasse A kann Objekte b aus sämtlichen Unterklassen B von A aufnehmen.
- Durch diese Zuweisung vergisst Java die Zugehörigkeit zu B, da Java alle Werte von x als Objekte der Klasse A behandelt.
- Mit dem Ausdruck `x instanceof B` können wir zur Laufzeit die Klassenzugehörigkeit von x testen;
- Sind wir uns sicher, dass x aus der Klasse B ist, können wir in diesen Typ casten.
- Ist der aktuelle Wert der Variablen x bei dem versuchten Cast tatsächlich ein Objekt (einer Unterklasse) der Klasse B, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine ↑Exception ausgelöst.

Was passiert hier eigentlich?

```
objB = (ClassB) objA;
```

Die Typinformationen der Objekte werden geprüft (zur Laufzeit) um sicherzustellen, dass `objA` ein `ClassB`-Objekt ist, d.h., dass es insbesondere `short b` enthält.

Hier gibt es einen Laufzeitfehler.



Fazit

- Eine Variable `x` einer Klasse A kann Objekte `b` aus sämtlichen Unterklassen B von A aufnehmen.
- Durch diese Zuweisung vergisst Java die Zugehörigkeit zu B, da Java alle Werte von `x` als Objekte der Klasse A behandelt.
- Mit dem Ausdruck `x instanceof` B können wir zur Laufzeit die Klassenzugehörigkeit von `x` testen;
- Sind wir uns sicher, dass `x` aus der Klasse B ist, können wir in diesen Typ casten.
- Ist der aktuelle Wert der Variablen `x` bei dem versuchten Cast tatsächlich ein Objekt (einer Unterklasse) der Klasse B, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine ↑Exception ausgelöst.

Beispiel — Listen

Wir definieren Liste für `Object` anstatt jeweils eine für `Rational`, `BankAccount`, etc.

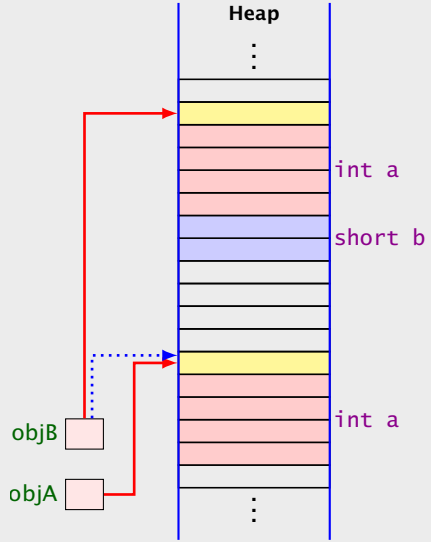
```
1  public class List {
2      public Object info;
3      public List next;
4      public List(Object x, List l) {
5          info = x;
6          next = l;
7      }
8      public void insert(Object x) {
9          next = new List(x,next);
10     }
11     public void delete() {
12         if (next != null) next = next.next;
13     }
14 // continued...
```


Was passiert hier eigentlich?

`objB = (ClassB) objA;`

Die Typinformationen der Objekte werden geprüft (zur Laufzeit) um sicherzustellen, dass `objA` ein `ClassB`-Objekt ist, d.h., dass es insbesondere `short b` enthält.

Hier gibt es einen Laufzeitfehler.



Beispiel — Listen

Wir definieren Liste für `Object` anstatt jeweils eine für `Rational`, `BankAccount`, etc.

```
1  public class List {
2      public Object info;
3      public List next;
4      public List(Object x, List l) {
5          info = x;
6          next = l;
7      }
8      public void insert(Object x) {
9          next = new List(x,next);
10     }
11     public void delete() {
12         if (next != null) next = next.next;
13     }
14 // continued...
```


Beispiel — Listen

```
14     public String toString() {
15         String result = "[" + info;
16         for (List t = next; t != null; t = t.next)
17             result = result+ ", " + t.info;
18         return result + "]";
19     }
20     ...
21 } // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `int`.
- Die `toString()`-Methode ruft implizit die (stets vorhandene) `toString()`-Methode der Listenelemente auf.

Beispiel — Listen

```
14     public String toString() {
15         String result = "[" + info;
16         for (List t = next; t != null; t = t.next)
17             result = result+ ", " + t.info;
18         return result + "]";
19     }
20     ...
21 } // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `int`.
- Die `toString()`-Methode ruft implizit die (stets vorhandene) `toString()`-Methode der Listenelemente auf.

Beispiel — Listen

Achtung:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = list.info;
5 System.out.println(x);
6 //...
```

liefert einen Compilerfehler. Der Variablen `x` dürfen nur Unterklassen von `Poly` zugewiesen werden.

Beispiel — Listen

Achtung:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = list.info;
5 System.out.println(x);
6 //...
```

liefert einen Compilerfehler. Der Variablen x dürfen nur Unterklassen von Poly zugewiesen werden.

Beispiel — Listen

Stattdessen:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = (Poly) list.info;
5 System.out.println(x);
6 //...
```

Das ist hässlich!!! Geht das nicht besser???

13.2 Generische Klassen

Idee:

- Java verfügt über generische Klassen...
- Anstatt das Attribut `info` als `Object` zu deklarieren, geben wir der Klasse einen Typ-Parameter `T` für `info` mit!
- Bei Anlegen eines Objekts der Klasse `List` bestimmen wir, welchen Typ `T` und damit `info` haben soll...

Beispiel — Listen

Stattdessen:

```
1 //...
2 Poly x = new Poly();
3 List list = new List(x);
4 x = (Poly) list.info;
5 System.out.println(x);
6 //...
```

Das ist hässlich!!! Geht das nicht besser???

13.2 Generische Klassen

Idee:

- Java verfügt über generische Klassen...
- Anstatt das Attribut `info` als `Object` zu deklarieren, geben wir der Klasse einen Typ-Parameter T für `info` mit!
- Bei Anlegen eines Objekts der Klasse `List` bestimmen wir, welchen Typ T und damit `info` haben soll...

Beispiel — Listen

```
1  public class List<T> {
2      public T info;
3      public List<T> next;
4      public List (T x, List<T> l) {
5          info = x;
6          next = l;
7      }
8      public void insert(T x) {
9          next = new List<T> (x, next);
10     }
11     public void delete() {
12         if (next != null) next = next.next;
13     }
14     //continued...
```

Beispiel — Listen

```
1  public class List<T> {
2      public T info;
3      public List<T> next;
4      public List (T x, List<T> l) {
5          info = x;
6          next = l;
7      }
8      public void insert(T x) {
9          next = new List<T> (x, next);
10     }
11     public void delete() {
12         if (next != null) next = next.next;
13     }
14     //continued...
```


Beispiel — Listen

```
15     public static void main (String [] args) {
16         List<Poly> list
17                 = new List<Poly> (new Poly(), null);
18         System.out.println(list.info.greeting());
19     }
20 } // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typparameter `T` = `Poly`.
- Deshalb ist `list.info` vom Typ `Poly`.
- Folglich ruft `list.info.greeting()` die entsprechende Methode der Klasse `Poly` auf.

Beispiel — Listen

```
15     public static void main (String [] args) {
16         List<Poly> list
17                 = new List<Poly> (new Poly(), null);
18         System.out.println(list.info.greeting());
19     }
20 } // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typparameter `T` = `Poly`.
- Deshalb ist `list.info` vom Typ `Poly`.
- Folglich ruft `list.info.greeting()` die entsprechende Methode der Klasse `Poly` auf.

Bemerkungen

- Die Typ-Parameter der Klasse dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden!!!
- Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:

  `A<S,T> extends B<T>` ist erlaubt.

  `A<S> extends B<S,T>` ist verboten.

- `Poly` ist eine Unterklasse von `Object`; aber `List<Poly>` ist keine Unterklasse von `List<Object>`!!!

Bemerkungen

- Die Typ-Parameter der Klasse dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden!!!
- Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:

  `A<S,T> extends B<T>` ist erlaubt.

  `A<S> extends B<S,T>` ist verboten.

- `Poly` ist eine Unterklasse von `Object`; aber `List<Poly>` ist keine Unterklasse von `List<Object>`!!!

Beispiel

```
1 List<Poly> l1 = new List<Poly>(new Poly(), null);
2 List<Object> l2 = l1;
3 l2.insert(new String("geht das?"));
```

```
1 Poly[] a1 = new Poly[100];
2 Object[] a2 = a1;
3 a2[0] = new String("geht das?");
```

Beispiel

```
1 List<Poly> l1 = new List<Poly>(new Poly(), null);
2 List<Object> l2 = l1;
3 l2.insert(new String("geht das?"));
```

```
1 Poly[] a1 = new Poly[100];
2 Object[] a2 = a1;
3 a2[0] = new String("geht das?");
```


Bemerkungen

- Für einen Typ-Parameter T kann man auch eine Oberklasse (oder ein Interface) angeben, das T auf jeden Fall erfüllen soll...

```
1  class Drawable {
2      void draw() {}
3  }
4  public class DrawableList<E extends Drawable> {
5      E element;
6      DrawableList<E> next;
7      void drawAll() {
8          element.draw();
9          if (next == null) return;
10         else next.drawAll();
11     }
12 }
```

13.3 Wrapper-Klassen

Problem

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.)

Ausweg

- Wickle die Werte eines Basis-Typs in ein Objekt ein!
  $\Longrightarrow$ Wrapper-Objekte aus Wrapper-Klassen.

Bemerkungen

- Für einen Typ-Parameter T kann man auch eine Oberklasse (oder ein Interface) angeben, das T auf jeden Fall erfüllen soll...

```
1  class Drawable {
2      void draw() {}
3  }
4  public class DrawableList<E extends Drawable> {
5      E element;
6      DrawableList<E> next;
7      void drawAll() {
8          element.draw();
9          if (next == null) return;
10         else next.drawAll();
11     }
12 }
```

13.3 Wrapper-Klassen

Problem

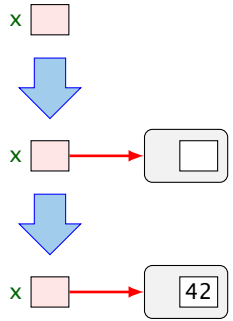
- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.)

Ausweg

- Wickle die Werte eines Basis-Typs in ein Objekt ein!
  $\Longrightarrow$ Wrapper-Objekte aus Wrapper-Klassen.

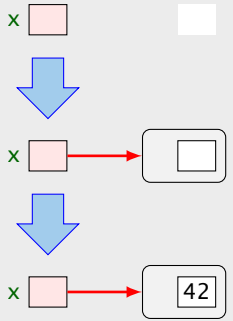
Beispiel

Die Zuweisung `Integer x = new Integer(42);` bewirkt:



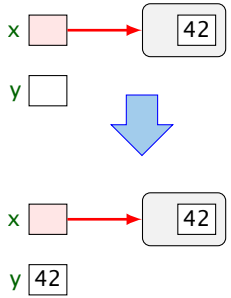
Beispiel

Die Zuweisung `Integer x = new Integer(42);` bewirkt:



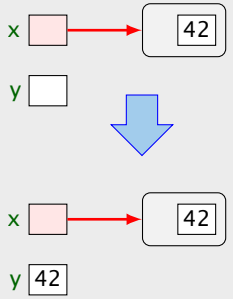
Beispiel

Eingewickelte Werte können auch wieder ausgewickelt werden.
Bei Zuweisung `int y = x;` erfolgt automatische Konvertierung:



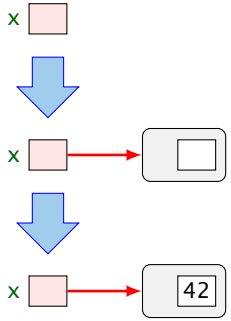
Beispiel

Eingewickelte Werte können auch wieder ausgewickelt werden.
Bei Zuweisung `int y = x;` erfolgt automatische Konvertierung:



Beispiel

Umgekehrt wird bei Zuweisung eines `int`-Werts an eine `Integer`-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Nützliches

Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

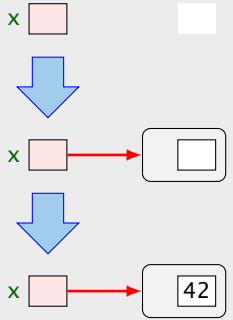
Beispiele:

- `public static int MIN_VALUE = -2147483648;` liefert den kleinsten `int`-Wert;
- `public static int MAX_VALUE = 2147483647;` liefert den größten `int`-Wert;
- `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem `String`-Objekt `s` die dargestellte Zahl — sofern `s` einen `int`-Wert darstellt.

  Andernfalls wird eine ↑Exception geworfen.

Beispiel

Umgekehrt wird bei Zuweisung eines `int`-Werts an eine `Integer`-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Nützliches

Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

Beispiele:

- `public static int MIN_VALUE = -2147483648;` liefert den kleinsten `int`-Wert;
- `public static int MAX_VALUE = 2147483647;` liefert den größten `int`-Wert;
- `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem `String`-Objekt `s` die dargestellte Zahl — sofern `s` einen `int`-Wert darstellt.

  Andernfalls wird eine ↑Exception geworfen.

Bemerkungen

- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

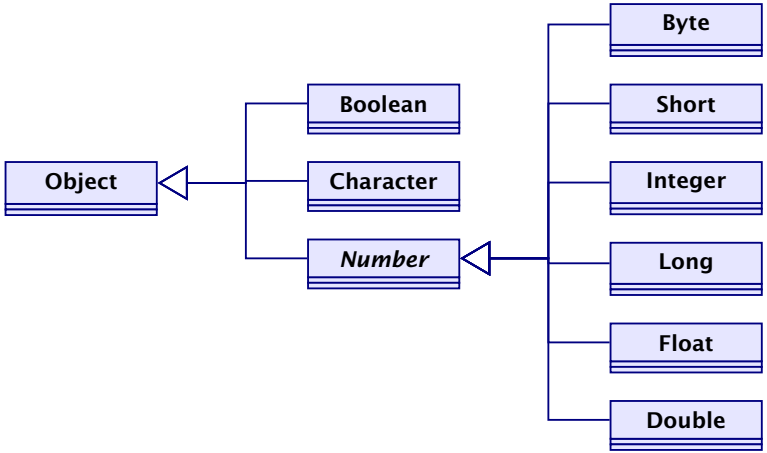
Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen...

Bemerkungen

- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen...

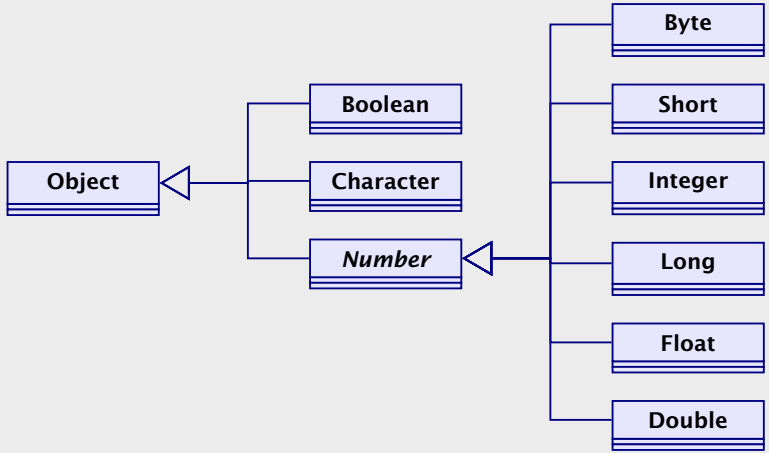
Wrapper-Klassen



Bemerkungen

- Sämtliche Wrapper-Klassen für Typen `type` (außer `char`) verfügen über
  - Konstruktoren aus Basiswerten bzw. `String`-Objekten;
  - eine statische Methode `type parseType(String s)`;
  - eine Methode `boolean equals(Object obj)` die auf Gleichheit testet (auch `Character`).
- Bis auf `Boolean` verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- `Character` enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln...
- Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- Diese Klasse ist ↑abstrakt d.h. man kann keine `Number`-Objekte anlegen.

Wrapper-Klassen



Spezielles

- `Double` und `Float` enthalten zusätzlich die Konstanten

  ```
  NEGATIVE_INFINITY  =  -1.0/0
  POSITIVE_INFINITY  =  +1.0/0
  NaN                =   0.0/0
  ```

- Zusätzlich gibt es die Tests
  - `public static boolean isInfinite(double v);`
    `public static boolean isNaN(double v);`
    (analog für `float`)
  - `public boolean isInfinite();`
    `public boolean isNaN();`

  mittels derer man auf (Un)Endlichkeit der Werte testen kann.

Bemerkungen

- Sämtliche Wrapper-Klassen für Typen `type` (außer `char`) verfügen über
  - Konstruktoren aus Basiswerten bzw. `String`-Objekten;
  - eine statische Methode `type` `parseType(String s)`;
  - eine Methode `boolean equals(Object obj)` die auf Gleichheit testet (auch `Character`).
- Bis auf `Boolean` verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- `Character` enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln...
- Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- Diese Klasse ist ↑abstrakt d.h. man kann keine `Number`-Objekte anlegen.

Spezielles

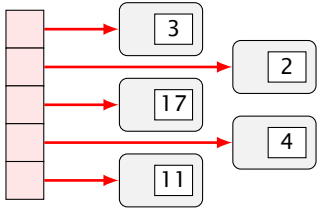
- `Double` und `Float` enthalten zusätzlich die Konstanten

  `NEGATIVE_INFINITY` = -1.0/0
  `POSITIVE_INFINITY` = +1.0/0
  `NaN` = 0.0/0

- Zusätzlich gibt es die Tests
  - `public static boolean isInfinite(double v);`
    `public static boolean isNaN(double v);`
    (analog für `float`)
  - `public boolean isInfinite();`
    `public boolean isNaN();`

  mittels derer man auf (Un)Endlichkeit der Werte testen kann.

Integer vs. Int



`Integer[]`

`int[]`

+ `Integer`s können in polymorphen Datenstrukturen hausen.
- Sie benötigen mehr als doppelt so viel Platz.
- Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten
  $\Longrightarrow$ schlechteres Cache-Verhalten.

14 Abstrakte Klassen, finale Klassen, Interfaces

- Eine abstrakte Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls abstrakt.
- Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden.
- Mit abstrakten Klassen können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

Auswertung von Ausdrücken

```
1  public abstract class Expression {
2      private int value;
3      private boolean evaluated = false;
4      public int getValue() {
5          if (!evaluated) {
6              value = evaluate();
7              evaluated = true;
8          }
9          return value;
10     }
11     abstract protected int evaluate();
12 } // end of class Expression
```

- Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` — immer mit einer anderen Implementierung.

14 Abstrakte Klassen, finale Klassen, Interfaces

- Eine abstrakte Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls abstrakt.
- Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden.
- Mit abstrakten Klassen können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

Abstrakte Methoden und Klassen

- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden — inklusive den Parametertypen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

Auswertung von Ausdrücken

```
1  public abstract class Expression {
2      private int value;
3      private boolean evaluated = false;
4      public int getValue() {
5          if (!evaluated) {
6              value = evaluate();
7              evaluated = true;
8          }
9          return value;
10     }
11     abstract protected int evaluate();
12 } // end of class Expression
```

- Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` — immer mit einer anderen Implementierung.

Abstrakte Methoden und Klassen

- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden — inklusive den Parametertypen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

Beispiel

- Die Methode `evaluate()` soll den Ausdruck auswerten.
- Die Methode `getValue()` speichert das Ergebnis in dem Attribut `value` ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

Beispiel für einen Ausdruck:

```
1 public final class Const extends Expression {
2     private int n;
3     public Const(int x) { n = x; }
4     protected int evaluate() {
5         return n;
6     } // end of evaluate()
7 } // end of class Const
```

Das Schlüsselwort final

- Der Ausdruck Const benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- Die Klasse ist als final deklariert.
- Zu als final deklarierten Klassen dürfen keine Unterklassen deklariert werden!!!
- Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als final deklariert werden...
- Statt ganzer Klassen können auch einzelne Variablen oder Methoden als final deklariert werden.
- Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden $\Longrightarrow$ Konstanten.

Beispiel

- Die Methode evaluate() soll den Ausdruck auswerten.
- Die Methode getValue() speichert das Ergebnis in dem Attribut value ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

Beispiel für einen Ausdruck:

```
1 public final class Const extends Expression {
2     private int n;
3     public Const(int x) { n = x; }
4     protected int evaluate() {
5         return n;
6     } // end of evaluate()
7 } // end of class Const
```

Andere Ausdrücke

```
1  public final class Add extends Expression {
2      private Expression left, right;
3      public Add(Expression l, Expression r) {
4          left = l; right = r;
5      }
6      protected int evaluate() {
7          return left.getValue() + right.getValue();
8      } // end of evaluate()
9  } // end of class Add
10 public final class Neg extends Expression {
11     private Expression arg;
12     public Neg(Expression a) { arg = a; }
13     protected int evaluate() { return -arg.getValue(); }
14 } // end of class Neg
```

Das Schlüsselwort `final`

- Der Ausdruck `Const` benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- Die Klasse ist als `final` deklariert.
- Zu als `final` deklarierten Klassen dürfen keine Unterklassen deklariert werden!!!
- Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als final deklariert werden...
- Statt ganzer Klassen können auch einzelne Variablen oder Methoden als `final` deklariert werden.
- Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden $\Longrightarrow$ Konstanten.

Andere Ausdrücke

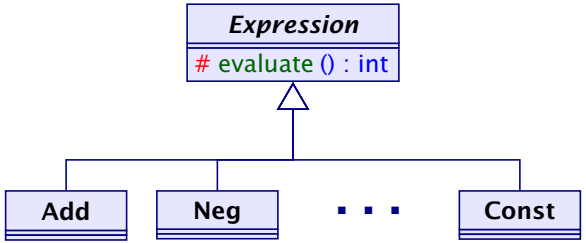
```
1  public final class Add extends Expression {
2      private Expression left, right;
3      public Add(Expression l, Expression r) {
4          left = l; right = r;
5      }
6      protected int evaluate() {
7          return left.getValue() + right.getValue();
8      } // end of evaluate()
9  } // end of class Add
10 public final class Neg extends Expression {
11     private Expression arg;
12     public Neg(Expression a) { arg = a; }
13     protected int evaluate() { return -arg.getValue(); }
14 } // end of class Neg
```


main()

```
1 public static void main(String[] args) {
2     Expression e = new Add (
3         new Neg (new Const(8)),
4         new Const(16));
5     System.out.println(e.getValue());
6 }
```

- Die Methode getValue() ruft eine Methode evaluate() sukzessive für jeden Teilausdruck von e auf.
- Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

Klassenhierarchie



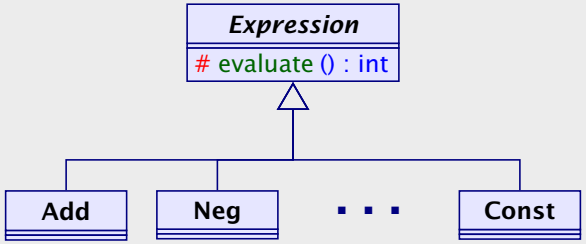
Leider (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren...

main()

```
1 public static void main(String[] args) {
2     Expression e = new Add (
3         new Neg (new Const(8)),
4         new Const(16));
5     System.out.println(e.getValue());
6 }
```

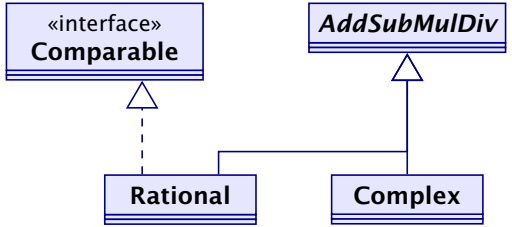
- Die Methode `getValue()` ruft eine Methode `evaluate()` sukzessive für jeden Teilausdruck von `e` auf.
- Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

Klassenhierarchie



Leider (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren...

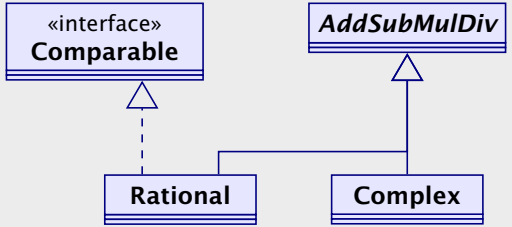
Beispiel



`AddSubMulDiv` = Objekte mit Operationen `add()`, `sub()` `mul()`, und `div()`

`Comparable` = Objekte, die eine `compareTo()`-Operation besitzen.

Beispiel



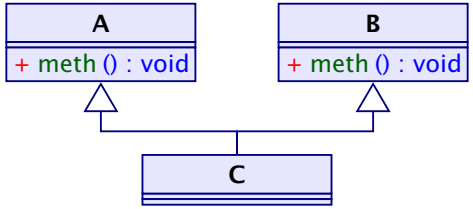
`AddSubMulDiv` = Objekte mit Operationen `add()`, `sub()` `mul()`, und `div()`

`Comparable` = Objekte, die eine `compareTo()`-Operation besitzen.

Mehrfachvererbung

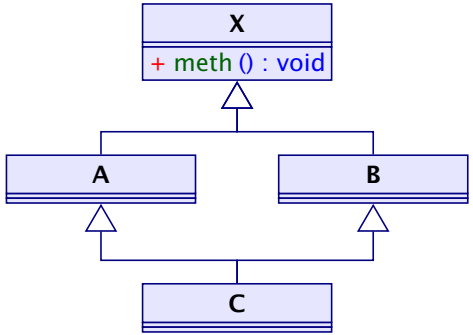
Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:

- Auf welche Klasse bezieht sich `super`?
- Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren?



Mehrfachvererbung

- Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren? Insbesondere

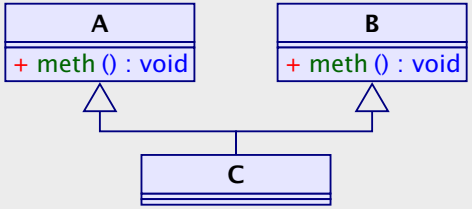


deadly diamond of death

Mehrfachvererbung

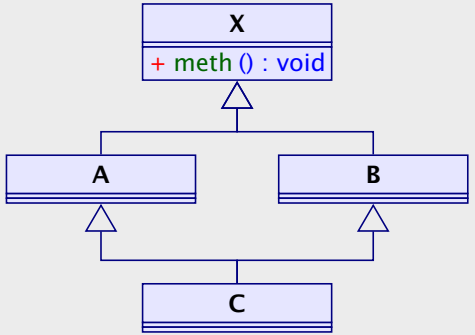
Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:

- Auf welche Klasse bezieht sich `super`?
- Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren?



Mehrfachvererbung

- Welche Objekt-Methode `meth()` ist gemeint, wenn mehrere Oberklassen `meth()` implementieren? Insbesondere



deadly diamond of death

Interfaces

- Kein Problem entsteht, wenn die Objekt-Methode `meth()` in allen Oberklassen abstrakt ist,
- oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt.

Ein Interface kann aufgefasst werden als eine abstrakte Klasse, wobei:

- alle Objekt-Methoden abstrakt sind;
- es keine Klassen-Methoden gibt;
- alle Variablen Konstanten sind.

Interfaces

- Kein Problem entsteht, wenn die Objekt-Methode `meth()` in allen Oberklassen abstrakt ist,
- oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt.

Ein Interface kann aufgefasst werden als eine abstrakte Klasse, wobei:

- alle Objekt-Methoden abstrakt sind;
- es keine Klassen-Methoden gibt;
- alle Variablen Konstanten sind.

Beispiel

```
1 public interface Comparable {
2     int compareTo(Object x);
3 }
```

- `Object` ist die gemeinsame Oberklasse aller Klassen.
- Methoden in Interfaces sind automatisch Objektmethoden und `public`.
- Es muss eine Obermenge der in Implementierungen geworfenen Exceptions angegeben werden.
- Evt. vorkommende Konstanten sind automatisch `public static`.

Beispiel

```
1 public interface Comparable {
2     int compareTo(Object x);
3 }
```

- `Object` ist die gemeinsame Oberklasse aller Klassen.
- Methoden in Interfaces sind automatisch Objektmethoden und `public`.
- Es muss eine Obermenge der in Implementierungen geworfenen Exceptions angegeben werden.
- Evt. vorkommende Konstanten sind automatisch `public static`.

Beispiel

```
1  public class Rational extends AddSubMulDiv
2                        implements Comparable {
3  private int zaehler, nenner;
4  public int compareTo(Object cmp) {
5      Rational fraction = (Rational) cmp;
6      long left  = (long)zaehler * (long)fraction.nenner;
7      long right = (long)nenner  * (long)fraction.zaehler;
8      return left == right ?  0:
9             left <  right ? -1:
10                            1;
11     } // end of compareTo
12     ...
13 } // end of class Rational
```

Beispiel

```
1  public class Rational extends AddSubMulDiv
2                        implements Comparable {
3  private int zaehler, nenner;
4  public int compareTo(Object cmp) {
5      Rational fraction = (Rational) cmp;
6      long left  = (long)zaehler * (long)fraction.nenner;
7      long right = (long)nenner  * (long)fraction.zaehler;
8      return left == right ?  0:
9             left <  right ? -1:
10                            1;
11     } // end of compareTo
12     ...
13 } // end of class Rational
```


Erläuterungen

- `class A extends B implements B1, B2,...,Bk {...}` gibt an, dass die Klasse `A` als Oberklasse `B` hat und zusätzlich die Interfaces `B1, B2,...,Bk` unterstützt, d.h. passende Objektmethoden zur Verfügung stellt.
- Java gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- Die Konstanten des Interface können in implementierenden Klassen direkt benutzt werden.
- Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

Erläuterungen

- `class A extends B implements B1, B2,...,Bk {...}` gibt an, dass die Klasse A als Oberklasse B hat und zusätzlich die Interfaces `B1, B2,...,Bk` unterstützt, d.h. passende Objektmethoden zur Verfügung stellt.
- Java gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- Die Konstanten des Interface können in implementierenden Klassen direkt benutzt werden.
- Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

Erläuterungen

- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten umdefinieren...
- Kommt eine Konstante gleichen Namens `const` in verschiedenen implementierten Interfaces A und B vor, kann man sie durch `A.const` und `B.const` unterscheiden.

Beispiel:

```
1 public interface Countable extends Comparable, Cloneable {
2     Countable next();
3     Countable prev();
4     int number();
5 }
```

Erläuterungen

- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten umdefinieren...
- Kommt eine Konstante gleichen Namens `const` in verschiedenen implementierten Interfaces A und B vor, kann man sie durch `A.const` und `B.const` unterscheiden.

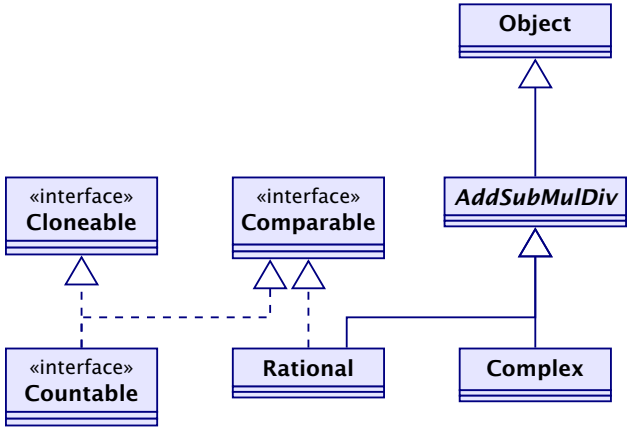
Beispiel:

```
1 public interface Countable extends Comparable, Cloneable {
2     Countable next();
3     Countable prev();
4     int number();
5 }
```


Erläuterungen

- Das Interface `Countable` umfasst die (beide vordefinierten) Interfaces `Comparable` und `Cloneable`.
- Das vordefinierte Interface `Cloneable` verlangt eine Objektmethode `public Object clone()` die eine Kopie des Objekts anlegt.
- Eine Klasse, die `Countable` implementiert, muss über die Objektmethoden `compareTo(), clone(), next(), prev()` und `number()` verfügen.

Übersicht



Erläuterungen

- Das Interface `Countable` umfasst die (beide vordefinierten) Interfaces `Comparable` und `Cloneable`.
- Das vordefinierte Interface `Cloneable` verlangt eine Objektmethode `public Object clone()` die eine Kopie des Objekts anlegt.
- Eine Klasse, die `Countable` implementiert, muss über die Objektmethoden `compareTo(), clone(), next(), prev()` und `number()` verfügen.

15 Fehlerobjekte: Werfen, Fangen, Behandeln

- Tritt während der Programm-Ausführung ein Fehler auf, wird die normale Programm-ausführung abgebrochen und ein Fehlerobjekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Fehlern zusammen.
- Ein Fehlerobjekt kann gefangen und geeignet behandelt werden.

15 Fehlerobjekte: Werfen, Fangen, Behandeln

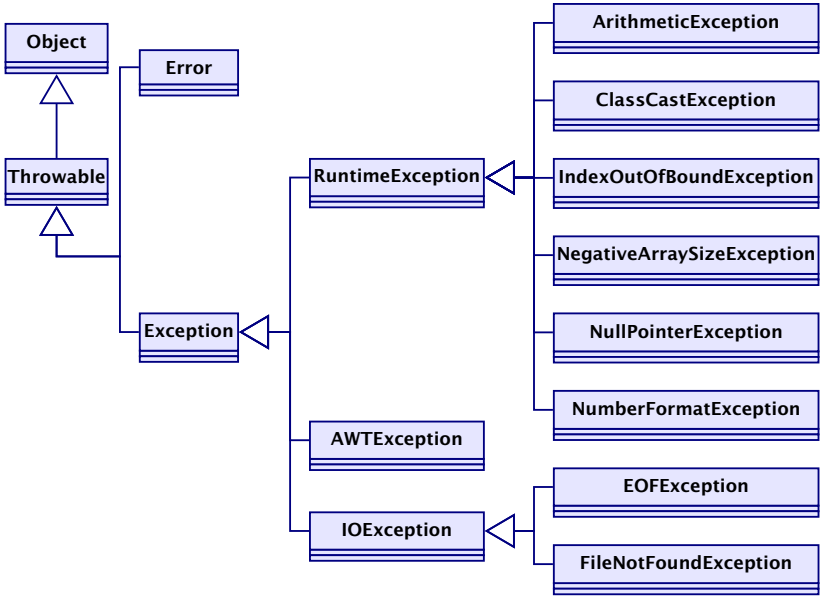
- Tritt während der Programm-Ausführung ein Fehler auf, wird die normale Programm-ausführung abgebrochen und ein Fehlerobjekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Fehlern zusammen.
- Ein Fehlerobjekt kann gefangen und geeignet behandelt werden.

Idee

Trennung von

- normalem Programm-Ablauf (der effizient und übersichtlich sein sollte); und
- Behandlung von Sonderfällen (wie illegalen Eingaben, falscher Benutzung, Sicherheitsattacken,...)

Fehlerklassen



Idee

Trennung von

- normalem Programm-Ablauf (der effizient und übersichtlich sein sollte); und
- Behandlung von Sonderfällen (wie illegalen Eingaben, falscher Benutzung, Sicherheitsattacken,...)

Fehlerklassen

Die direkten Unterklassen von `Throwable` sind:

- `Error` — für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- `Exception` — für bewältigbare Fehler oder Ausnahmen.

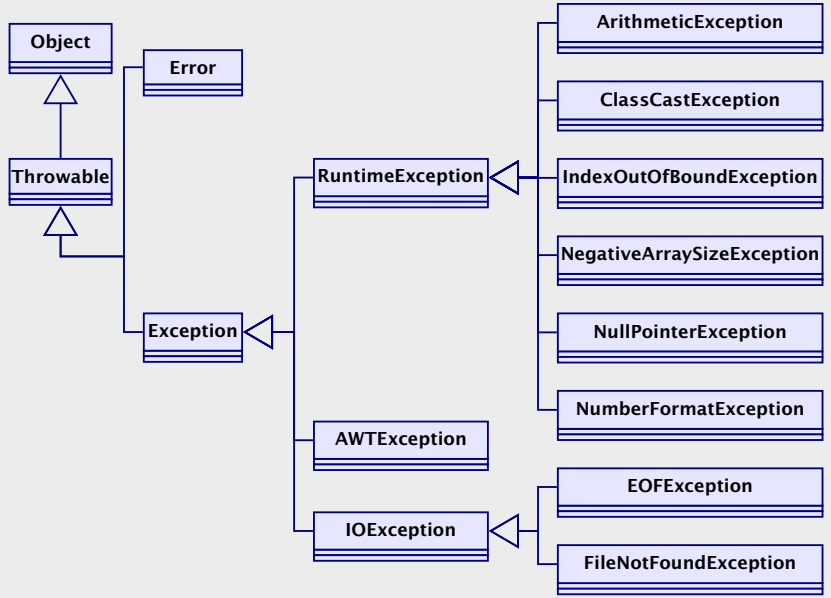
unchecked exception

Ausnahmen der Klasse `Error` und `RuntimeException` müssen nicht im Methodenkopf deklariert werden.

checked exception

Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode aufgelistet werden!!!

Fehlerklassen



Fehlerklassen

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programmausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten...
- Sie kann, muss aber nicht abgefangen werden.

Arten der Fehlerbehandlung:

- Ignorieren;
- Abfangen und Behandeln dort, wo sie entstehen;
- Abfangen und Behandeln an einer anderen Stelle.

Fehlerklassen

Die direkten Unterklassen von `Throwable` sind:

- `Error` — für fatale Fehler, die zur Beendigung des gesamten Programms führen, und
- `Exception` — für bewältigbare Fehler oder Ausnahmen.

unchecked exception

Ausnahmen der Klasse `Error` und `RuntimeException` müssen nicht im Methodenkopf deklariert werden.

checked exception

Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode aufgelistet werden!!!

Fehlerklassen

- Die Unterklasse `RuntimeException` der Klasse `Exception` fasst die bei normaler Programmausführung evt. auftretenden Ausnahmen zusammen.
- Eine `RuntimeException` kann jederzeit auftreten...
- Sie kann, muss aber nicht abgefangen werden.

Arten der Fehlerbehandlung:

- Ignorieren;
- Abfangen und Behandeln dort, wo sie entstehen;
- Abfangen und Behandeln an einer anderen Stelle.

Fehlerbehandlung

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programmausführung ab.

Beispiel:

```
1 public class Zero {
2     public static void main(String[]
          args) {
3         int x = 10;
4         int y = 0;
5         System.out.println(x/y);
6     } // end of main()
7 } // end of class Zero
```

Fehlermeldung

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: „/ by zero“.
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Call-Stack.

Fehlerbehandlung

Tritt ein Fehler auf und wird nicht behandelt, bricht die Programmausführung ab.

Beispiel:

```
1 public class Zero {
2     public static void main(String[]
          args) {
3         int x = 10;
4         int y = 0;
5         System.out.println(x/y);
6     } // end of main()
7 } // end of class Zero
```

Fehlerbehandlung

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: NumberFormatException

```
1 public class Adding extends MiniJava {
2     public static void main(String[] args) {
3         int x = getInt("1. Zahl:\t");
4         int y = getInt("2. Zahl:\t");
5         write("Summe:\t\t" + (x+y));
6     } // end of main()
7     public static int getInt(String str) {
8 //continued...
```


Fehlermeldung

Das Programm bricht wegen Division durch `(int)0` ab und liefert die Fehler-Meldung:

```
Exception in  thread "main" java.lang.ArithmeticException: / by zero
at Zero.main(Compiled Code)
```

Die Fehlermeldung besteht aus drei Teilen:

1. der ↑Thread, in dem der Fehler auftrat;
2. `System.err.println(toString());` d.h. dem Namen der Fehlerklasse, gefolgt von einer Fehlermeldung, die die Objekt-Methode `getMessage()` liefert, hier: „/ by zero“.
3. `printStackTrace(System.err);` d.h. der Funktion, in der der Fehler auftrat, genauer: der Angabe sämtlicher Aufrufe im Call-Stack.

Fehlerbehandlung

Soll die Programm-Ausführung nicht beendet werden, muss der Fehler abgefangen werden.

Beispiel: NumberFormatException

```
1 public class Adding extends MiniJava {
2     public static void main(String[] args) {
3         int x = getInt("1. Zahl:\t");
4         int y = getInt("2. Zahl:\t");
5         write("Summe:\t\t" + (x+y));
6     } // end of main()
7     public static int getInt(String str) {
8 //continued...
```


Erläuterungen

- Das Programm liest zwei int-Werte ein und addiert sie.
- Bei der Eingabe können möglicherweise Fehler auftreten:
  - ...weil die Eingabe keine syntaktisch korrekte Zahl ist;
  - ...weil sonstige unvorhersehbare Ereignisse eintreffen.
- Die Behandlung dieser Fehler ist in der Funktion getInt() verborgen...

Fehlerbehandlung

```
 9          String s;
10          while (true) {
11              try {
12                  s = readString(str);
13                  return Integer.parseInt(s);
14              } catch (NumberFormatException e) {
15                  System.out.println(
16                       "Falsche Eingabe! ...");
17              } catch (IOException e) {
18                  System.out.println(
19                       "Eingabeproblem: Ende ...");
20                  System.exit(0);
21              }
22          } // end of while
23      } // end of getInt()
24  } // end of class Adding
```


Erläuterungen

- Das Programm liest zwei `int`-Werte ein und addiert sie.
- Bei der Eingabe können möglicherweise Fehler auftreten:
  - ... weil die Eingabe keine syntaktisch korrekte Zahl ist;
  - ... weil sonstige unvorhersehbare Ereignisse eintreffen.
- Die Behandlung dieser Fehler ist in der Funktion `getInt()` verborgen...

Fehlerbehandlung

```
 9          String s;
10          while (true) {
11              try {
12                  s = readString(str);
13                  return Integer.parseInt(s);
14              } catch (NumberFormatException e) {
15                  System.out.println(
16                       "Falsche Eingabe! ...");
17              } catch (IOException e) {
18                  System.out.println(
19                       "Eingabeproblem: Ende ...");
20                  System.exit(0);
21              }
22          } // end of while
23      } // end of getInt()
24  } // end of class Adding
```


Beispielablauf

```
> java Adding
1. Zahl: abc
Falsche Eingabe! ...
1. Zahl: 0.3
Falsche Eingabe! ...
1. Zahl: 17
2. Zahl: 25
Summe: 42
```

Ausnahmebehandlung

- Ein Exception-Handler besteht aus einem `try`-Block `try{`*ss*`}`, in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge *ss* kein Fehlerobjekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

Beispielablauf

```
> java Adding
1. Zahl: abc
Falsche Eingabe! ...
1. Zahl: 0.3
Falsche Eingabe! ...
1. Zahl: 17
2. Zahl: 25
Summe: 42
```

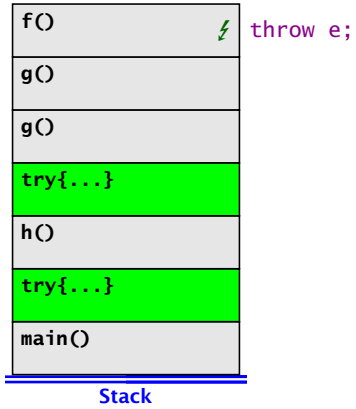
Ausnahmebehandlung

- Ein Exception-Handler besteht aus einem `try`-Block `try{`*ss*`}`, in dem der Fehler möglicherweise auftritt; gefolgt von einer oder mehreren `catch`-Regeln.
- Wird bei der Ausführung der Statement-Folge *ss* kein Fehlerobjekt erzeugt, fährt die Programm-Ausführung direkt hinter dem Handler fort.
- Wird eine Exception ausgelöst, durchsucht der Handler mithilfe des geworfenen Fehler-Objekts sequentiell die `catch`-Regeln.

Ausnahmebehandlung

- Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}` wobei `Exc` eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehlerobjekt aus (einer Unterklasse) von `Exc` stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

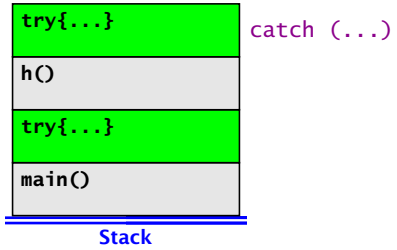
Was passiert auf dem Stack?



Ausnahmebehandlung

- Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}` wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehlerobjekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

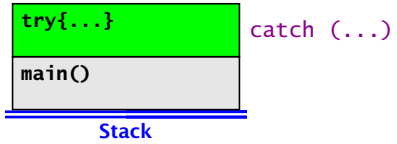
Was passiert auf dem Stack?



Ausnahmebehandlung

- Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}` wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehlerobjekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

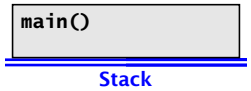
Was passiert auf dem Stack?



Ausnahmebehandlung

- Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}` wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehlerobjekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

Was passiert auf dem Stack?



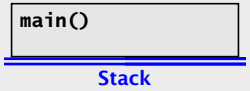
Ausnahmebehandlung

- Jede `catch`-Regel ist von der Form: `catch(Exc e) {...}` wobei Exc eine Klasse von Fehlern angibt und `e` ein formaler Parameter ist, an den das Fehler-Objekt gebunden wird.
- Eine Regel ist anwendbar, sofern das Fehlerobjekt aus (einer Unterklasse) von Exc stammt.
- Die erste `catch`-Regel, die anwendbar ist, wird angewendet. Dann wird der Handler verlassen.
- Ist keine `catch`-Regel anwendbar, wird der Fehler propagiert.

Ausnahmebehandlung

- Auslösen eines Fehlers verlässt abrupt die aktuelle Berechnung.
- Damit das Programm trotz Auftretens des Fehlers in einem geordneten Zustand bleibt, ist oft Aufräumarbeit erforderlich – z.B. das Schließen von I/O-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

Was passiert auf dem Stack?



Ausnahmebehandlung

- Auslösen eines Fehlers verlässt abrupt die aktuelle Berechnung.
- Damit das Programm trotz Auftretens des Fehlers in einem geordneten Zustand bleibt, ist oft Aufräumarbeit erforderlich – z.B. das Schließen von I/O-Strömen.
- Dazu dient `finally { ss }` nach einem `try`-Statement.

Ausnahmebehandlung

- Die Folge `ss` von Statements wird auf jeden Fall ausgeführt.
- Wird kein Fehler im `try`-Block geworfen, wird sie im Anschluss an den `try`-Block ausgeführt.
- Wird ein Fehler geworfen und mit einer `catch`-Regel behandelt, wird sie nach dem Block der `catch`-Regel ausgeführt.
- Wird der Fehler von keiner `catch`-Regel behandelt, wird `ss` ausgeführt, und dann der Fehler weitergereicht.

Beispiel NullPointerException

```
1  public class Kill {
2      public static void kill() {
3          Object x = null; x.hashCode ();
4      }
5      public static void main(String[] args) {
6          try { kill();
7          } catch (ClassCastException b) {
8              System.out.println("Falsche Klasse!!!");
9          } finally {
10             System.out.println("Leider nix gefangen
                   ...");
11         }
12     } // end of main()
13 } // end of class Kill
```


Ausnahmebehandlung

- Die Folge ss von Statements wird auf jeden Fall ausgeführt.
- Wird kein Fehler im try-Block geworfen, wird sie im Anschluss an den try-Block ausgeführt.
- Wird ein Fehler geworfen und mit einer catch-Regel behandelt, wird sie nach dem Block der catch-Regel ausgeführt.
- Wird der Fehler von keiner catch-Regel behandelt, wird ss ausgeführt, und dann der Fehler weitergereicht.

Beispiel NullPointerException

```
1  public class Kill {
2      public static void kill() {
3          Object x = null; x.hashCode ();
4      }
5      public static void main(String[] args) {
6          try { kill();
7          } catch (ClassCastException b) {
8              System.out.println("Falsche Klasse!!!");
9          } finally {
10             System.out.println("Leider nix gefangen
                   ...");
11         }
12     } // end of main()
13 } // end of class Kill
```


Resultat:

```
> java Kill
Leider nix gefangen ...
Exception in thread "main" java.lang.NullPointerException
        at Kill.kill(Compiled Code)
        at Kill.main(Compiled Code)
```

Selbstdefinierte Fehler

Exceptions können auch

- selbst definiert und
- selbst geworfen werden.

Beispiel:

```
1 public class Killed extends Exception {
2     Killed() {}
3     Killed(String s) {super(s);}
4 } // end of class Killed
5 public class Kill {
6     public static void kill() throws Killed {
7         throw new Killed();
8     }
9 // continued...
```

Resultat:

```
> java Kill
Leider nix gefangen ...
Exception in thread "main" java.lang.NullPointerException
        at Kill.kill(Compiled Code)
        at Kill.main(Compiled Code)
```

Beispiel

```
10      public static void main(String[] args) {
11          try {
12              kill();
13          } catch (RuntimeException r) {
14              System.out.println("RunTimeException "+ r);
15          } catch (Killed b) {
16              System.out.println("Killed It!");
17              System.out.println(b);
18              System.out.println(b.getMessage());
19          }
20      } // end of main
21  } // end of class Kill
```

Selbstdefinierte Fehler

Exceptions können auch

- selbst definiert und
- selbst geworfen werden.

Beispiel:

```
1  public class Killed extends Exception {
2      Killed() {}
3      Killed(String s) {super(s);}
4  } // end of class Killed
5  public class Kill {
6      public static void kill() throws Killed {
7          throw new Killed();
8      }
9  // continued...
```

Selbstdefinierte Fehler

- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden!
- Die Klasse `Exception` verfügt über die Konstruktoren
  `public Exception();`
  `public Exception(String str);`
  (`str` ist die evt. auszugebende Fehlermeldung).
- `throw exc` löst den Fehler `exc` aus — sofern sich der Ausdruck `exc` zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen.
- Ausgabe:

```
Killed It!
Killed
Null
```

Beispiel

```
10      public static void main(String[] args) {
11          try {
12              kill();
13          } catch (RuntimeException r) {
14              System.out.println("RunTimeException "+ r);
15          } catch (Killed b) {
16              System.out.println("Killed It!");
17              System.out.println(b);
18              System.out.println(b.getMessage());
19          }
20      } // end of main
21  } // end of class Kill
```

Fazit

- Fehler in Java sind Objekte und können vom Programm selbst behandelt werden.
- `try ... catch ... finally` gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Selbstdefinierte Fehler

- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden!
- Die Klasse `Exception` verfügt über die Konstruktoren
  ```
  public Exception();
  public Exception(String str);
  ```
  (`str` ist die evt. auszugebende Fehlermeldung).
- `throw` exc löst den Fehler exc aus — sofern sich der Ausdruck exc zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen.
- Ausgabe:
  ```
  Killed It!
  Killed
  Null
  ```

Checked Exceptions

Vorteile:

- Man sieht sofort welche Exceptions eine Funktion werfen kann.
- Der Compiler „überprüft“ ob man die Exceptions behandelt.

Mögliche Nachteile:

- Skalierbarkeit: Wenn man viele API-Funktionen aufruft muß man eventuell sehr viele Exceptions angeben, die man werfen könnte.
- Exceptions können Details der Implementierung verraten (e.g. `SQLException`). D.h. es könnte schwierig sein die Implementierung einer API-Funktion später zu ändern.

Fazit

- Fehler in Java sind Objekte und können vom Programm selbst behandelt werden.
- `try ... catch ... finally` gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Exceptions – Hinweise

- Exceptions fangen ist teuer. D.h. man sollte Exceptions nur für außergewöhnliche Situationen nutzen. Nicht zur Ablaufkontrolle.
- Fehler sollten dort behandelt werden, wo sie auftreten.
- Es ist besser spezifischere Fehler zu fangen als allgemeine — z.B. mit `catch(Exception e) {}`
- Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Eine API-Funktion sollte Exceptions werfen, die der Abstraktion der Funktion angemessen sind. (Also `getUser()` sollte z.B. `UserNotFoundException` werfen, anstatt `SQLException`.)
- Programmierfehler (z.B. falsche Verwendung einer Klasse) sollten durch `RunTimeException` signalisiert werden.
- Checked Exceptions sollte man nur verwenden, wenn der Aufrufer sie sinnvoll behandeln kann.

Checked Exceptions

Vorteile:

- Man sieht sofort welche Exceptions eine Funktion werfen kann.
- Der Compiler „überprüft“ ob man die Exceptions behandelt.

Mögliche Nachteile:

- Skalierbarkeit: Wenn man viele API-Funktionen aufruft muß man eventuell sehr viele Exceptions angeben, die man werfen könnte.
- Exceptions können Details der Implementierung verraten (e.g. `SQLException`). D.h. es könnte schwierig sein die Implementierung einer API-Funktion später zu ändern.

Exception Safety

Es gibt verschiedene Garantien, die eine Objektmethode bzgl. Exceptions erfüllen kann:

Basic Guarantee
Nach einer Exception erfüllt das Objekt alle seine Invarianten. Keine Leaks.

Häufig nicht sehr hilfreich. Das Objekt kann sich beliebig verändert haben; das Gute ist, dass wir das Objekt noch löschen können.

Strong Guarantee
Im Fall einer Exception hat sich das Objekt nicht verändert. (Transaktionales Verhalten; entweder funktioniert alles, oder keine Änderung wird durchgeführt).

No-throw Guarantee
Die Funktion wirft keine Exceptions.

Exceptions – Hinweise

- Exceptions fangen ist teuer. D.h. man sollte Exceptions nur für außergewöhnliche Situationen nutzen. Nicht zur Ablaufkontrolle.
- Fehler sollten dort behandelt werden, wo sie auftreten.
- Es ist besser spezifischere Fehler zu fangen als allgemeine — z.B. mit `catch(Exception e) {}`
- Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Eine API-Funktion sollte Exceptions werfen, die der Abstraktion der Funktion angemessen sind. (Also `getUser()` sollte z.B. `UserNotFoundException` werfen, anstatt `SQLException`.)
- Programmierfehler (z.B. falsche Verwendung einer Klasse) sollten durch `RunTimeException` signalisiert werden.
- Checked Exceptions sollte man nur verwenden, wenn der Aufrufer sie sinnvoll behandeln kann.

Threads – Einführung

- Die Ausführung eines Java-Programms besteht in Wahrheit nicht aus einem, sondern mehreren parallel laufenden Threads.
- Ein Thread ist ein sequentieller Ausführungsstrang.
- Der Aufruf eines Programms startet einen Thread `main`, der die Methode `main()` des Programms ausführt.
- Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die Garbage Collection.
- Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programmausführung zur Verfügung stellen.

Threads – Einführung

- Die Ausführung eines Java-Programms besteht in Wahrheit nicht aus einem, sondern mehreren parallel laufenden Threads.
- Ein Thread ist ein sequentieller Ausführungsstrang.
- Der Aufruf eines Programms startet einen Thread `main`, der die Methode `main()` des Programms ausführt.
- Ein weiterer Thread, den das Laufzeitsystem parallel startet, ist die Garbage Collection.
- Die Garbage Collection soll mittlerweile nicht mehr erreichbare Objekte beseitigen und den von ihnen belegten Speicherplatz der weiteren Programmausführung zur Verfügung stellen.

Threads – Anwendungen

- Mehrere Threads sind auch nützlich, um
  - ...mehrere Eingabe-Quellen zu überwachen (z.B. Maus, Tastatur) ↑Graphik;
  - ...während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ...die Rechenkraft mehrerer Prozessoren auszunutzen.
- Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- Dazu stellt Java die Klasse `Thread` und das Interface `Runnable` bereit.

Threads – Anwendungen

- Mehrere Threads sind auch nützlich, um
  - ...mehrere Eingabe-Quellen zu überwachen (z.B. Maus, Tastatur) ↑Graphik;
  - ...während der Blockierung einer Aufgabe etwas anderes Sinnvolles erledigen zu können;
  - ...die Rechenkraft mehrerer Prozessoren auszunutzen.
- Neue Threads können deshalb vom Programm selbst erzeugt und gestartet werden.
- Dazu stellt Java die Klasse `Thread` und das Interface `Runnable` bereit.

Version A

```
1  public class MyThread extends Thread {
2      public void hello(String s) {
3          System.out.println(s);
4      }
5      public void run() {
6          hello("I'm running ...");
7      } // end of run()
8      public static void main(String[] args) {
9          MyThread t = new MyThread();
10         t.start();
11         System.out.println("Thread has been started
               ...");
12     } // end of main()
13 } // end of class MyThread
```

Version A

```
1  public class MyThread extends Thread {
2      public void hello(String s) {
3          System.out.println(s);
4      }
5      public void run() {
6          hello("I'm running ...");
7      } // end of run()
8      public static void main(String[] args) {
9          MyThread t = new MyThread();
10         t.start();
11         System.out.println("Thread has been started
               ...");
12     } // end of main()
13 } // end of class MyThread
```


Erläuterungen

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode

  `public void run();` implementieren.
- Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:

  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objektmethode `run()` für `t` wird angestoßen;
  3. die eigene Programmausführung wird hinter dem Aufruf fortgesetzt.

Version B

```
1  public class MyRunnable implements Runnable {
2      public void hello(String s) {
3          System.out.println(s);
4      }
5      public void run() {
6          hello("I'm running ...");
7      } // end of run()
8      public static void main(String[] args) {
9          Thread t = new Thread(new MyRunnable());
10         t.start();
11         System.out.println("Thread has been started
               ...");
12     } // end of main()
13 } // end of class MyRunnable
```


Erläuterungen

- Neue Threads werden für Objekte aus (Unter-) Klassen der Klasse `Thread` angelegt.
- Jede Unterklasse von `Thread` sollte die Objekt-Methode

  `public void run();` implementieren.
- Ist `t` ein `Thread`-Objekt, dann bewirkt der Aufruf `t.start();` das folgende:

  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objektmethode `run()` für `t` wird angestoßen;
  3. die eigene Programmausführung wird hinter dem Aufruf fortgesetzt.

Version B

```
1  public class MyRunnable implements Runnable {
2      public void hello(String s) {
3          System.out.println(s);
4      }
5      public void run() {
6          hello("I'm running ...");
7      } // end of run()
8      public static void main(String[] args) {
9          Thread t = new Thread(new MyRunnable());
10         t.start();
11         System.out.println("Thread has been started
               ...");
12     } // end of main()
13 } // end of class MyRunnable
```


Erläuterungen

- Auch das Interface `Runnable` verlangt die Implementierung einer Objektmethode `public void run();`
- `public Thread(Runnable obj);` legt für ein `Runnable`-Objekt `obj` ein `Thread`-Objekt an.
- Ist `t` das `Thread`-Objekt für das `Runnable` `obj`, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für `obj` wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Erläuterungen

- Auch das Interface `Runnable` verlangt die Implementierung einer Objektmethode `public void run();`
- `public Thread(Runnable obj);` legt für ein `Runnable`-Objekt `obj` ein `Thread`-Objekt an.
- Ist `t` das `Thread`-Objekt für das `Runnable` `obj`, dann bewirkt der Aufruf `t.start();` das folgende:
  1. ein neuer Thread wird initialisiert;
  2. die (parallele) Ausführung der Objekt-Methode `run()` für `obj` wird angestoßen;
  3. die eigene Programm-Ausführung wird hinter dem Aufruf fortgesetzt.

Mögliche Ausführungen

Entweder

```
Thread has been started ...
I'm running ...
```

oder

```
I'm running ...
Thread has been started ...
```

Mögliche Ausführungen

Entweder

```
Thread has been started ...
I'm running ...
```

oder

```
I'm running ...
Thread has been started ...
```


Scheduling

- Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- Im Allgemeinen gibt es mehr Threads als CPUs.
- Der Scheduler verwaltet die verfügbaren CPUs und teilt sie den Threads zu.
- Bei verschiedenen Programmläufen kann diese Zuteilung verschieden aussehen!!!
- Es gibt verschiedene Strategien, nach denen sich Scheduler richten können (↑Betriebssysteme). Z.B.:
  - Zeitscheibenverfahren
  - Naives Verfahren

Scheduling

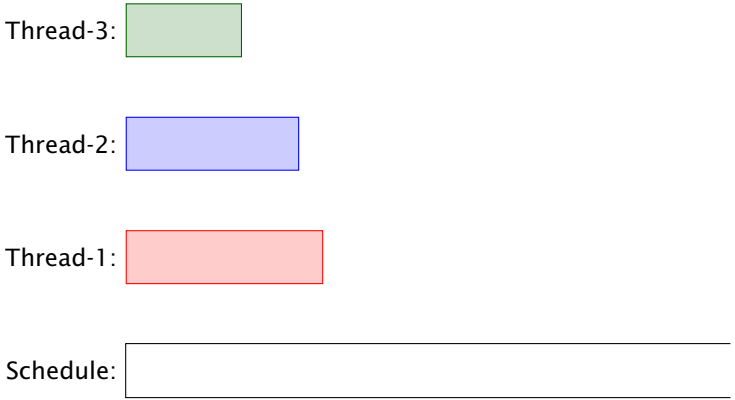
- Ein Thread kann nur eine Operation ausführen, wenn ihm ein Prozessor (CPU) zur Ausführung zugeteilt worden ist.
- Im Allgemeinen gibt es mehr Threads als CPUs.
- Der Scheduler verwaltet die verfügbaren CPUs und teilt sie den Threads zu.
- Bei verschiedenen Programmläufen kann diese Zuteilung verschieden aussehen!!!
- Es gibt verschiedene Strategien, nach denen sich Scheduler richten können (↑Betriebssysteme). Z.B.:
  - Zeitscheibenverfahren
  - Naives Verfahren

Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

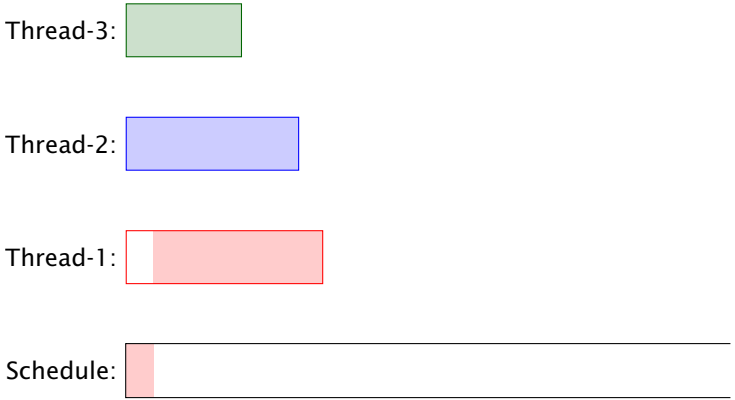


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

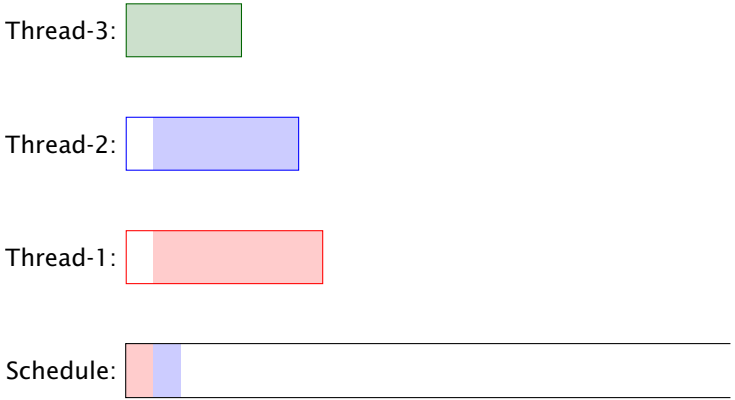


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

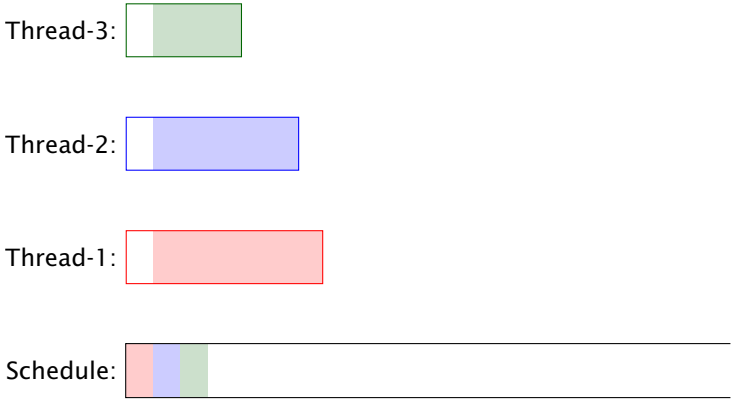


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

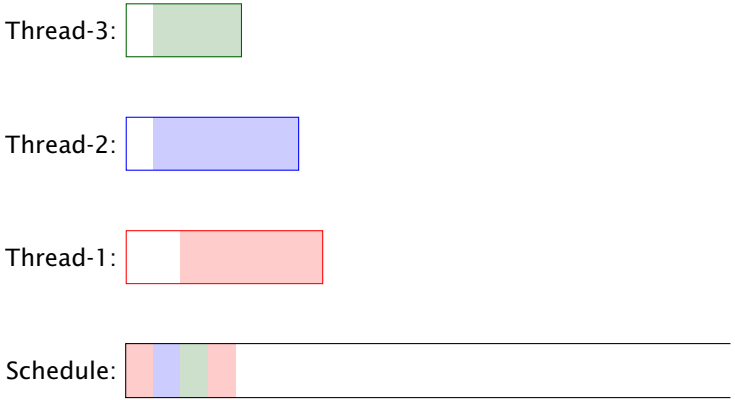


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

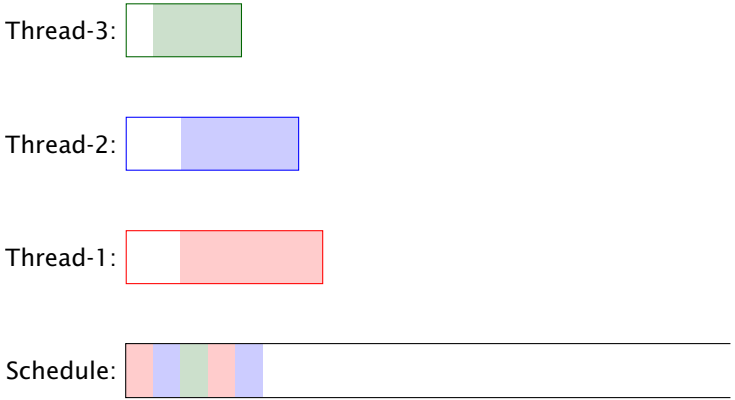


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren



Zeitscheibenverfahren

Strategie

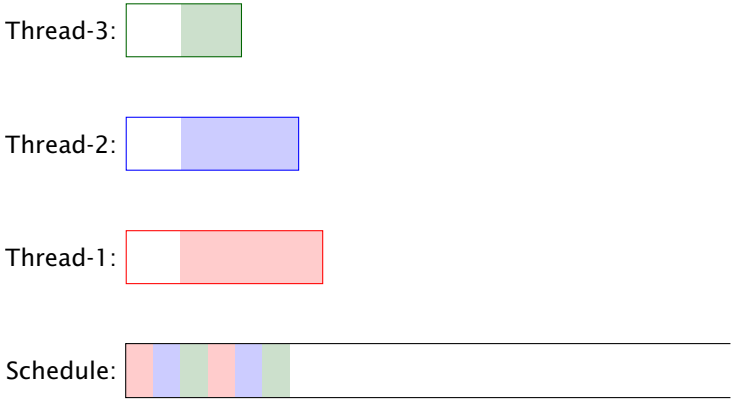
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Zeitscheibenverfahren

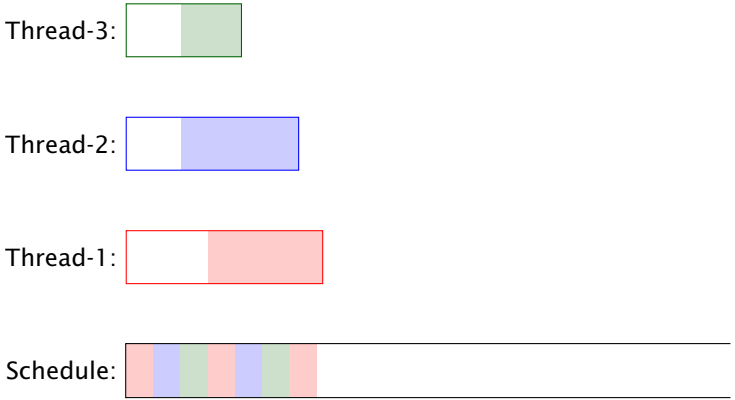
Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren



Beispiel: Zeitscheibenverfahren

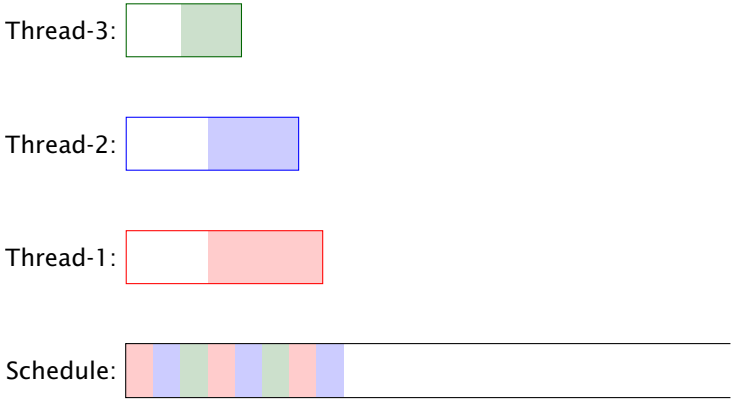


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

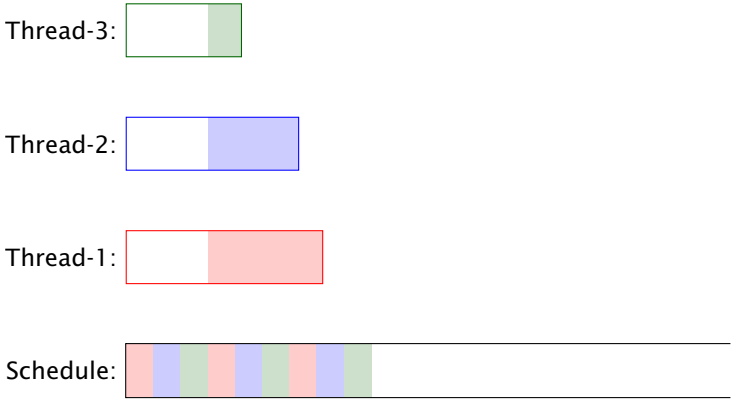


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

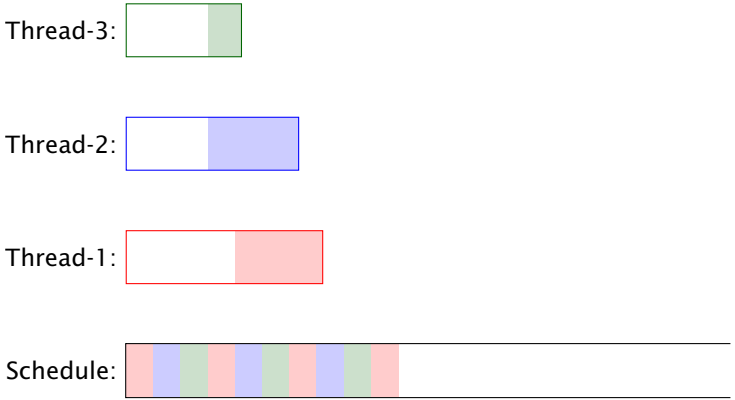


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

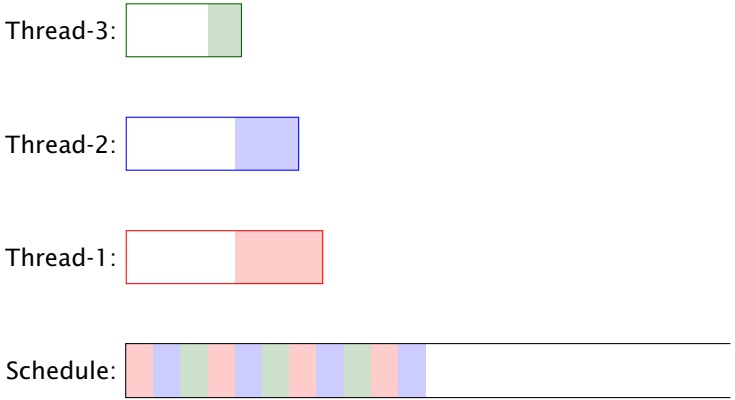


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

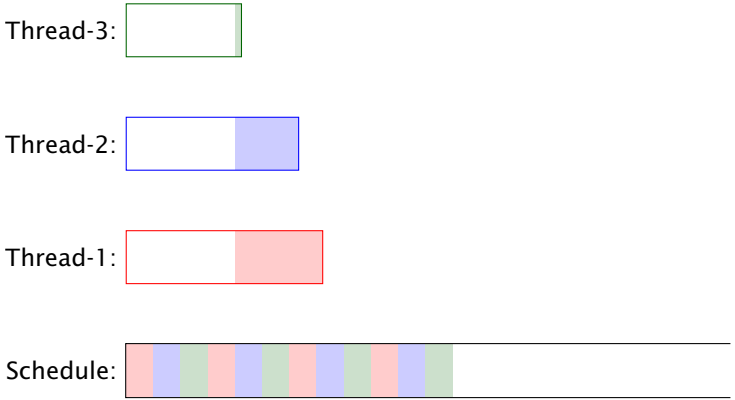


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

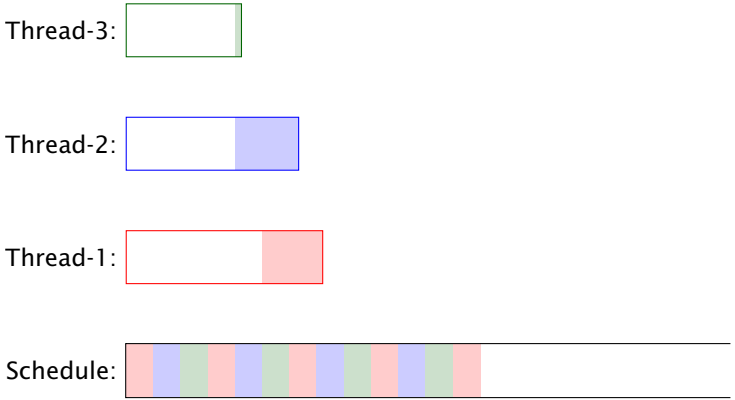


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

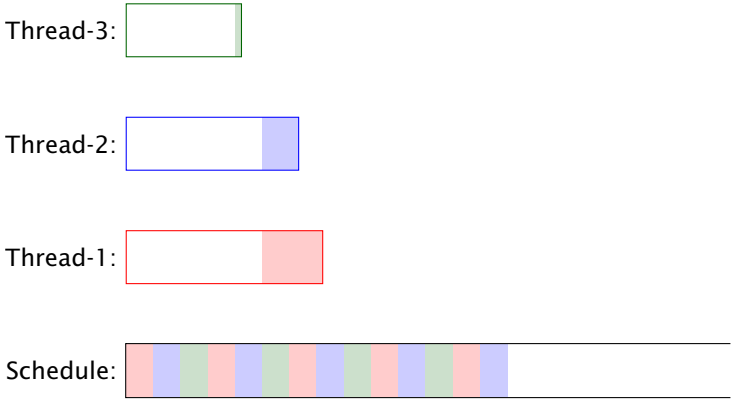


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

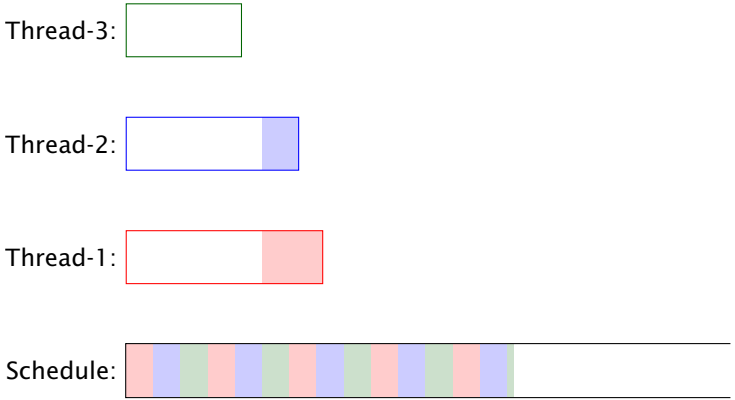


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren



Zeitscheibenverfahren

Strategie

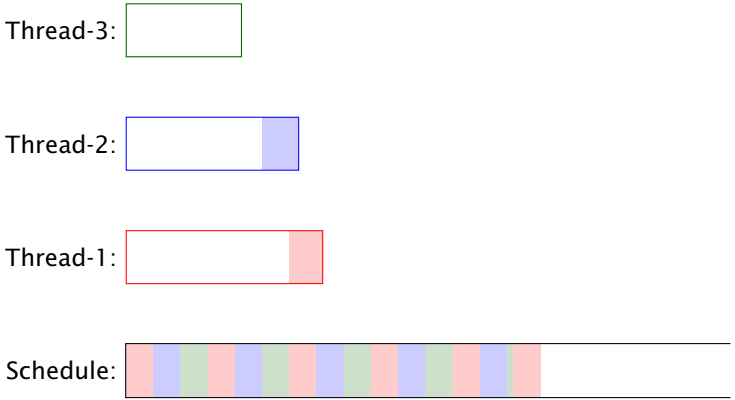
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Zeitscheibenverfahren

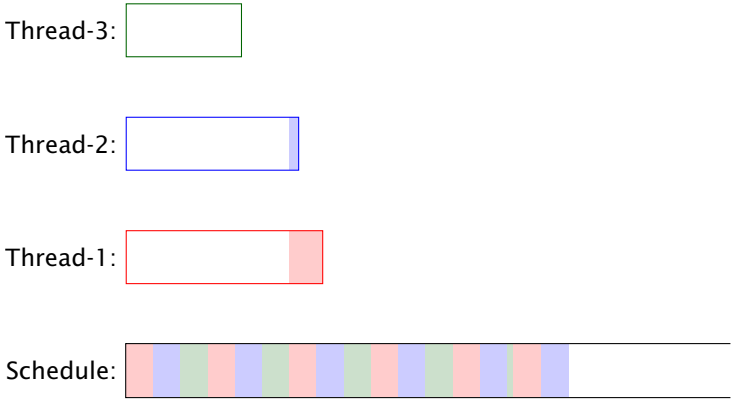
Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren



Beispiel: Zeitscheibenverfahren

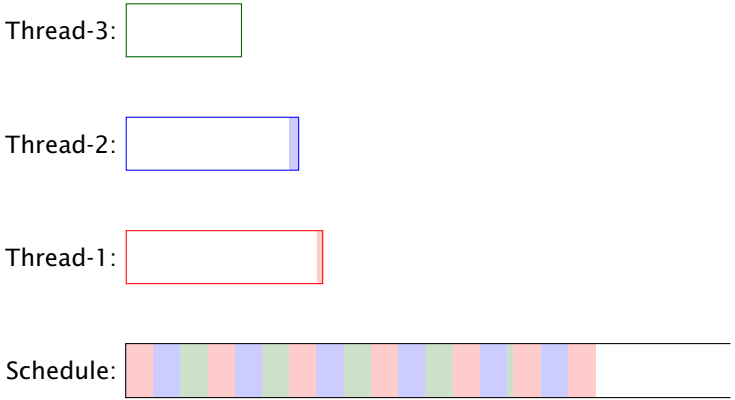


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

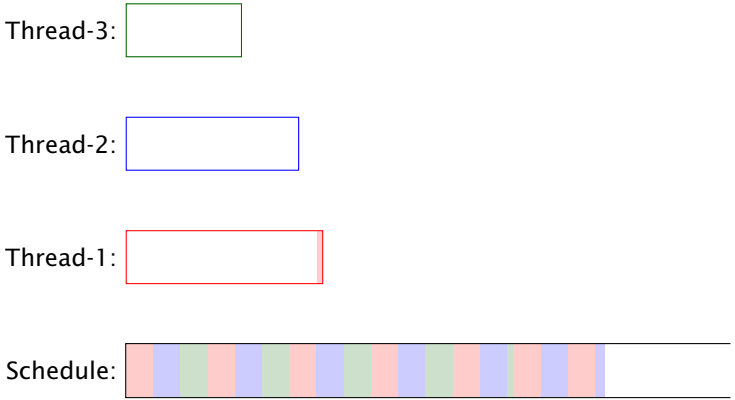


Zeitscheibenverfahren

Strategie

- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren



Zeitscheibenverfahren

Strategie

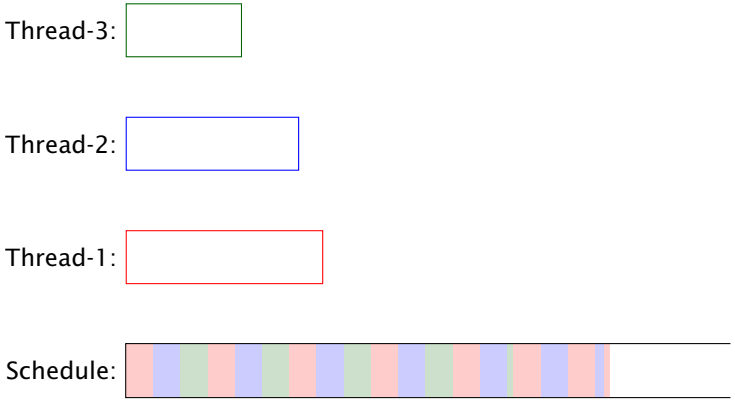
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Zeitscheibenverfahren

Strategie

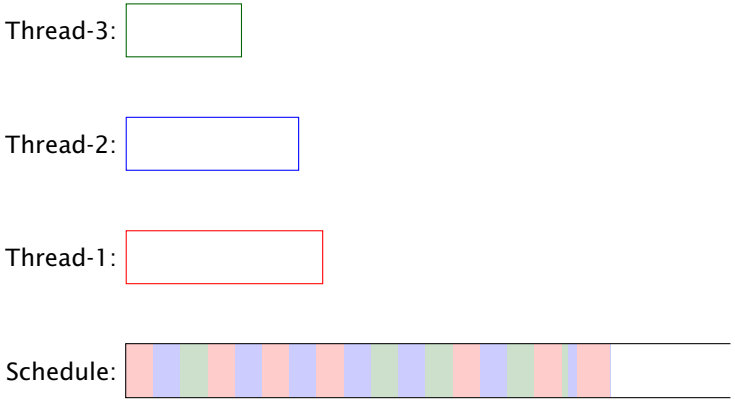
- Ein Thread erhält eine CPU nur für eine bestimmte Zeitspanne (Time Slice), in der er rechnen darf.
- Danach wird er unterbrochen. Dann darf ein anderer.

Beispiel: Zeitscheibenverfahren

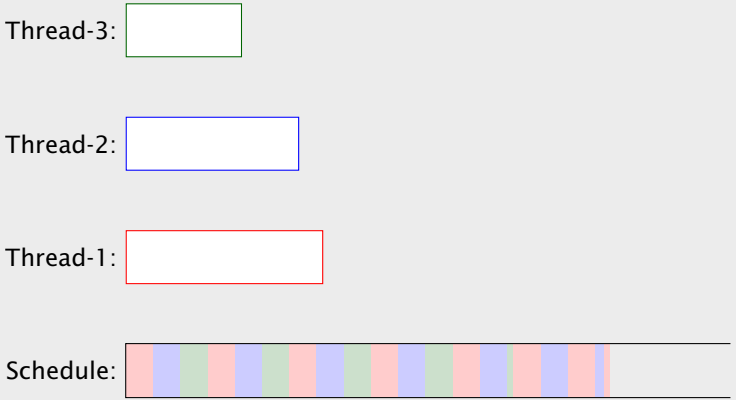


Beispiel: Zeitscheibenverfahren

Eine andere Ausführung:



Beispiel: Zeitscheibenverfahren

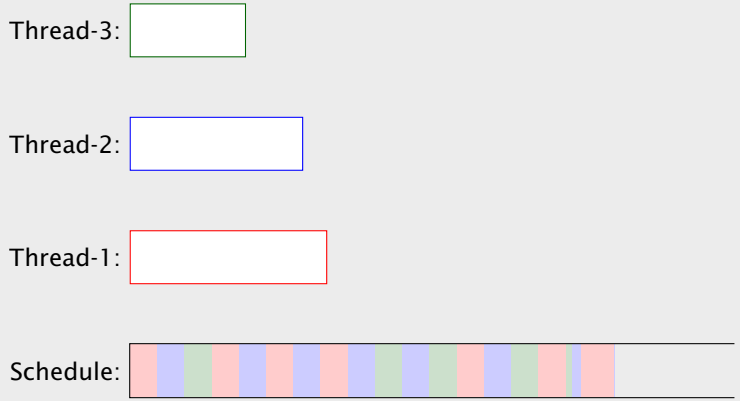


Erläuterungen – Zeitscheibenverfahren

- Ein Zeitscheiben-Scheduler versucht, jeden Thread fair zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen — egal, welche Threads sonst noch Rechenzeit beanspruchen.
- Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice.
- Für den Programmierer sieht es so aus, als ob sämtliche Threads „echt“ parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt.

Beispiel: Zeitscheibenverfahren

Eine andere Ausführung:



Erläuterungen – Zeitscheibenverfahren

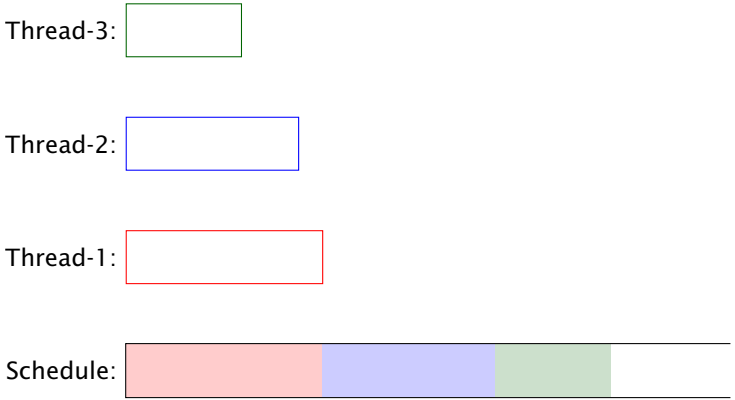
- Ein Zeitscheiben-Scheduler versucht, jeden Thread fair zu behandeln, d.h. ab und zu Rechenzeit zuzuordnen — egal, welche Threads sonst noch Rechenzeit beanspruchen.
- Kein Thread hat jedoch Anspruch auf einen bestimmten Time-Slice.
- Für den Programmierer sieht es so aus, als ob sämtliche Threads „echt“ parallel ausgeführt werden, d.h. jeder über eine eigene CPU verfügt.

Naives Scheduling

Strategie

- Erhält ein Thread eine CPU, darf er laufen, so lange er will...
- Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten...

Beispiel – Naives Scheduling



Naives Scheduling

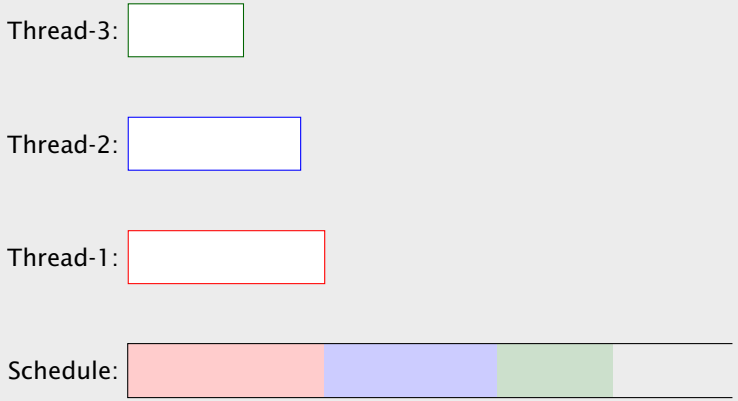
Strategie

- Erhält ein Thread eine CPU, darf er laufen, so lange er will...
- Gibt er die CPU wieder frei, darf ein anderer Thread arbeiten...

Beispiel

```
1  public class Start extends Thread {
2      public void run() {
3          System.out.println("I'm running...");
4          while (true);
5      }
6      public static void main(String[] args) {
7          (new Start()).start();
8          (new Start()).start();
9          (new Start()).start();
10         System.out.println("main is running...");
11         while (true);
12     }
13 } // end of class Start
```


Beispiel – Naives Scheduling



Beispiel

```
1  public class Start extends Thread {
2      public void run() {
3          System.out.println("I'm running...");
4          while (true);
5      }
6      public static void main(String[] args) {
7          (new Start()).start();
8          (new Start()).start();
9          (new Start()).start();
10         System.out.println("main is running...");
11         while (true);
12     }
13 } // end of class Start
```


Beispiel

Ausgabe (bei naivem Scheduling)

```
main is running...
```

Weil `main` nie fertig wird, erhalten die anderen Threads keine Chance, sie verhungern.

Faires Scheduling mit Zeitscheibenverfahren würde z.B. liefern:

```
I'm running...
main is running...
I'm running...
I'm running...
```

Implementierung in Java

- Java legt nicht fest, wie intelligent der Scheduler ist.
- Die aktuelle Implementierung unterstützt faires Scheduling.
- Programme sollten aber für jeden Scheduler das gleiche Verhalten zeigen. Das heißt:
- . . . Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
- . . . Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.

  Achtung: Threadwechsel ist teuer!!!
- Dazu verfügt jeder Thread über einen Zustand, der bei der Vergabe von Rechenzeit beücksichtigt wird.

Beispiel

Ausgabe (bei naivem Scheduling)

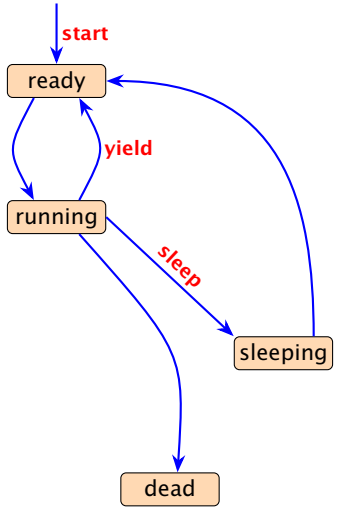
```
main is running...
```

Weil `main` nie fertig wird, erhalten die anderen Threads keine Chance, sie verhungern.

Faires Scheduling mit Zeitscheibenverfahren würde z.B. liefern:

```
I'm running...
main is running...
I'm running...
I'm running...
```

Threadzustände

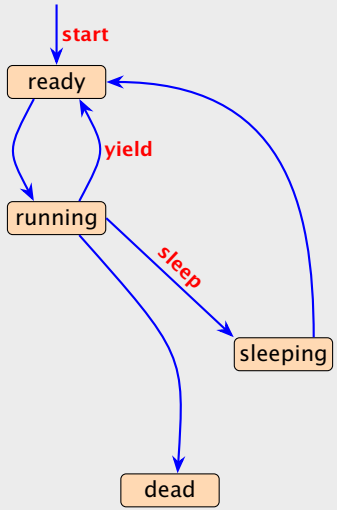


Implementierung in Java

- Java legt nicht fest, wie intelligent der Scheduler ist.
- Die aktuelle Implementierung unterstützt faires Scheduling.
- Programme sollten aber für jeden Scheduler das gleiche Verhalten zeigen. Das heißt:
- . . . Threads, die aktuell nichts sinnvolles zu tun haben, z.B. weil sie auf Verstreichen der Zeit oder besseres Wetter warten, sollten stets ihre CPU anderen Threads zur Verfügung stellen.
- . . . Selbst wenn Threads etwas Vernünftiges tun, sollten sie ab und zu andere Threads laufen lassen.

  Achtung: Threadwechsel ist teuer!!!
- Dazu verfügt jeder Thread über einen Zustand, der bei der Vergabe von Rechenzeit beücksichtigt wird.

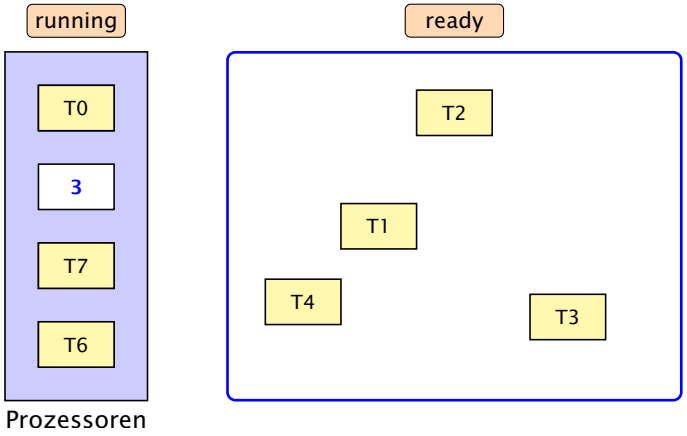
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

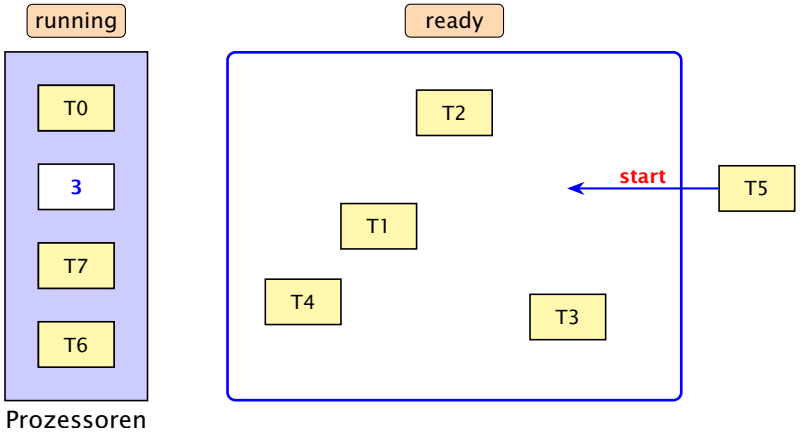
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

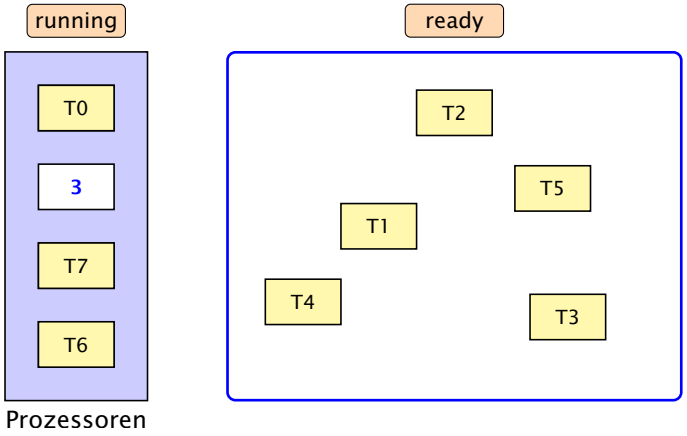
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

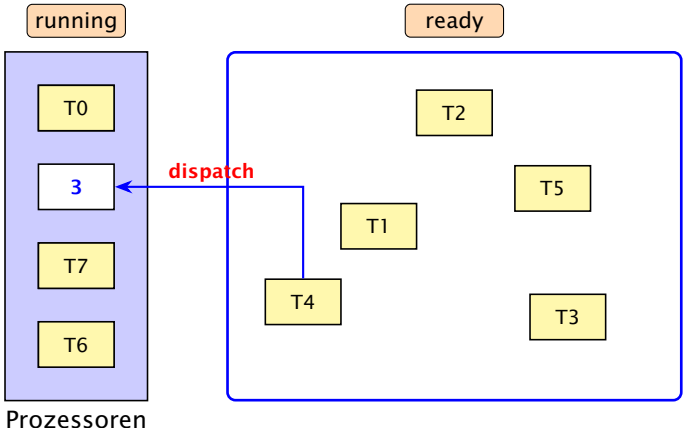
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

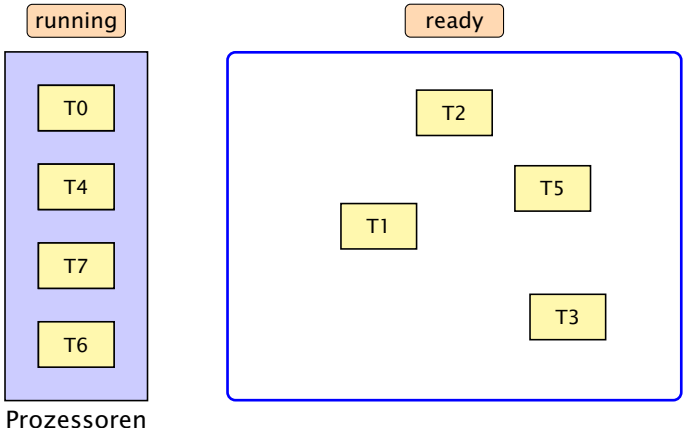
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

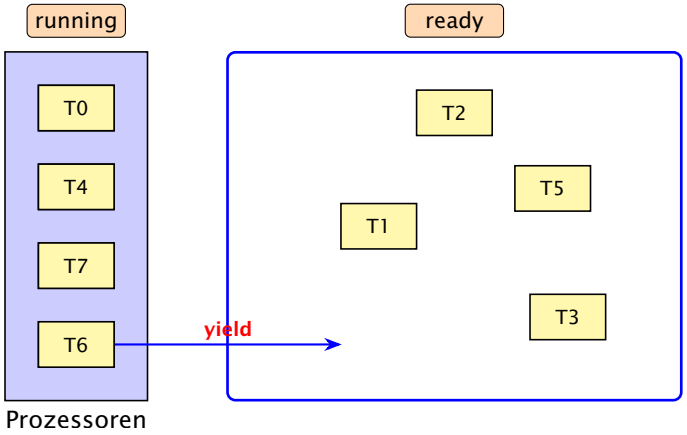
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

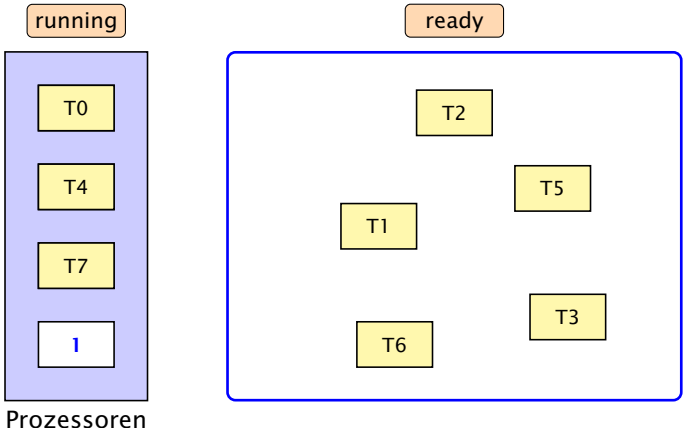
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

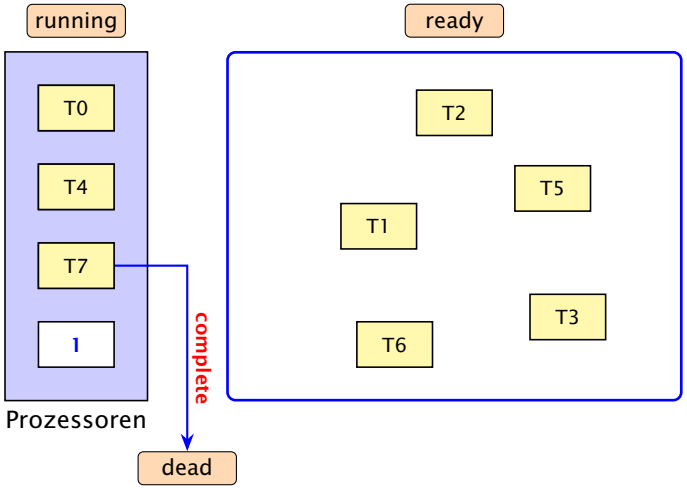
Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

Threadzustände



Threadzustände

- `public void start();` legt einen neuen Thread an, setzt den Zustand auf ready und übergibt damit den Thread dem Scheduler zur Ausführung.
- Der Scheduler ordnet den Threads, die im Zustand ready sind, Prozessoren zu („dispatching“). Aktuell laufende Threads haben den Zustand running.
- `public static void yield();` setzt den aktuellen Zustand zurück auf ready und unterbricht damit die aktuelle Programmausführung. Andere ausführbare Threads erhalten die Gelegenheit zur Ausführung.
- `public static void sleep(int msec) throws InterruptedException;` legt den aktuellen Thread für `msec` Millisekunden schlafen, indem der Thread in den Zustand sleeping wechselt.

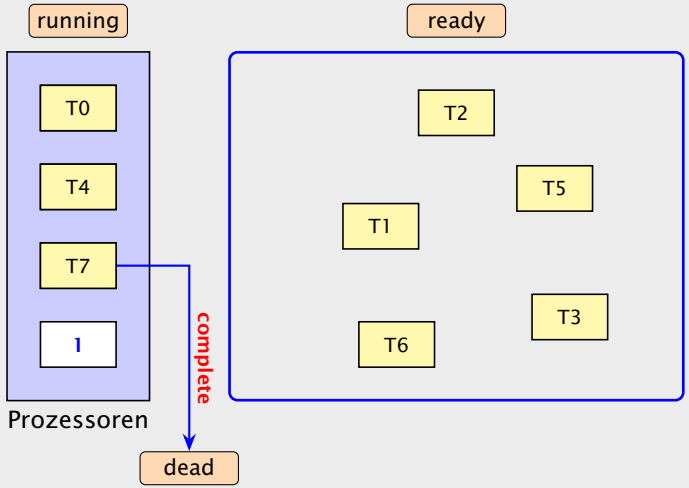
16.1 Futures

- Die Berechnung eines Zwischenergebnisses kann lange dauern.
- Während dieser Berechnung kann möglicherweise etwas anderes Sinnvolles berechnet werden.

Idee:

- Berechne das Zwischenergebnis in einem eigenen Thread.
- Greife auf den Wert erst zu, wenn sich der Thread beendet hat.

Threadzustände



16.1 Futures

Eine Future startet die Berechnung eines Werts, auf den später zugegriffen wird. Das generische Interface

```
public interface Callable<T> {
    T call () throws Exception;
}
```

aus java.util.concurrency beschreibt Klassen, für deren Objekte ein Wert vom Typ T berechnet werden kann.

```
1 public class Future<T> implements Runnable {
2     private T value = null;
3     private Exception exc = null;
4     private Callable<T> work;
5     private Thread task;
6 // continued...
```

16.1 Futures

- Die Berechnung eines Zwischenergebnisses kann lange dauern.
- Während dieser Berechnung kann möglicherweise etwas anderes Sinnvolles berechnet werden.

Idee:

- Berechne das Zwischenergebnis in einem eigenen Thread.
- Greife auf den Wert erst zu, wenn sich der Thread beendet hat.

Implementierung

```
7     public Future<T>(Callable<T> w) {
8         work = w;
9         task = new Thread (this);
10        task.start();
11    }
12    public void run() {
13        try {value = work.call();}
14        catch (Exception e) { exc = e;}
15    }
16    public T get() throws Exception {
17        task.join();
18        if (exc != null) throw exc;
19        return value;
20    }
21 }
```


16.1 Futures

Eine Future startet die Berechnung eines Werts, auf den später zugegriffen wird. Das generische Interface

```
public interface Callable<T> {
    T call () throws Exception;
}
```

aus java.util.concurrency beschreibt Klassen, für deren Objekte ein Wert vom Typ T berechnet werden kann.

```
1 public class Future<T> implements Runnable {
2     private T value = null;
3     private Exception exc = null;
4     private Callable<T> work;
5     private Thread task;
6 // continued...
```

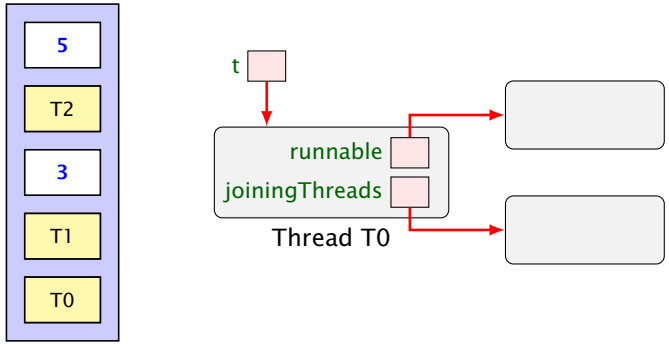
Implementierung

```
 7     public Future<T>(Callable<T> w) {
 8         work = w;
 9         task = new Thread (this);
10         task.start();
11     }
12     public void run() {
13         try {value = work.call();}
14         catch (Exception e) { exc = e;}
15     }
16     public T get() throws Exception {
17         task.join();
18         if (exc != null) throw exc;
19         return value;
20     }
21 }
```


Erläuterungen

- Der Konstruktor erhält ein `Callable`-Objekt.
- Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread`...
- Dann liefert `get()` den berechneten Wert zurück — falls keine Exception geworfen wurde. Andernfalls wird die Exception `exc` erneut geworfen.

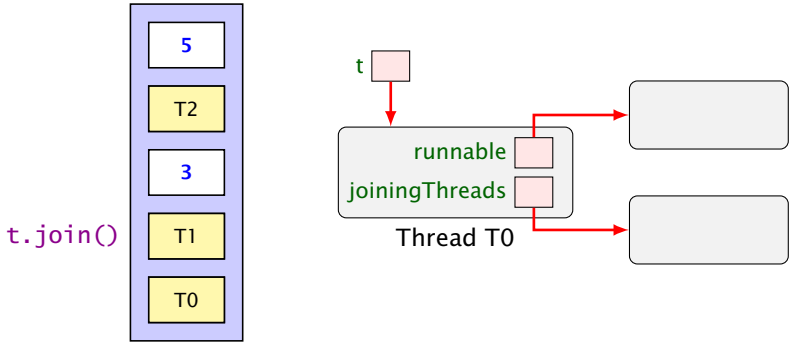
Die Join-Operation



Erläuterungen

- Der Konstruktor erhält ein `Callable`-Objekt.
- Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread`...
- Dann liefert `get()` den berechneten Wert zurück — falls keine Exception geworfen wurde. Andernfalls wird die Exception `exc` erneut geworfen.

Die Join-Operation



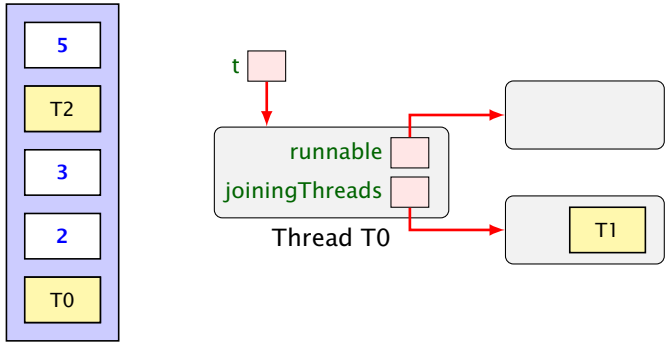
Erläuterungen

- Der Konstruktor erhält ein `Callable`-Objekt.
- Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread`...
- Dann liefert `get()` den berechneten Wert zurück — falls keine Exception geworfen wurde. Andernfalls wird die Exception `exc` erneut geworfen.

Erläuterungen

- Der Konstruktor erhält ein `Callable`-Objekt.
- Die Methode `run()` ruft für dieses Objekt die Methode `call()` auf und speichert deren Ergebnis in dem Attribut `value` — bzw. eine geworfene Exception in `exc` ab.
- Der Konstruktor legt ein Thread-Objekt für die Future an und startet diesen Thread, der dann `run()` ausführt.
- Die Methode `get()` wartet auf Beendigung des Threads. Dazu verwendet sie die Objekt-Methode `public final void join() throws InterruptedException` der Klasse `Thread`...
- Dann liefert `get()` den berechneten Wert zurück — falls keine Exception geworfen wurde. Andernfalls wird die Exception `exc` erneut geworfen.

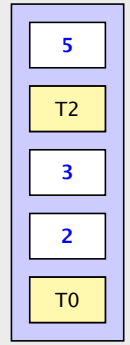
Die Join-Operation

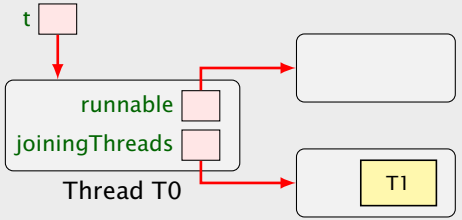


Erläuterungen

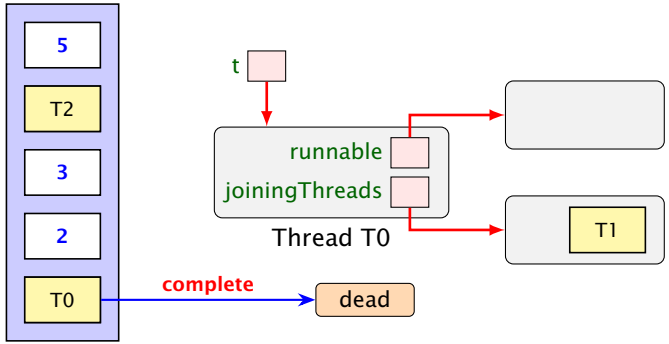
- Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- Dabei gehen sie konzeptuell in einen Zustand joining über und werden aus der Menge der ausführbaren Threads entfernt.
- Beendet sich ein Thread, werden alle Threads, die auf ihn warteten, wieder aktiviert...

Die Join-Operation





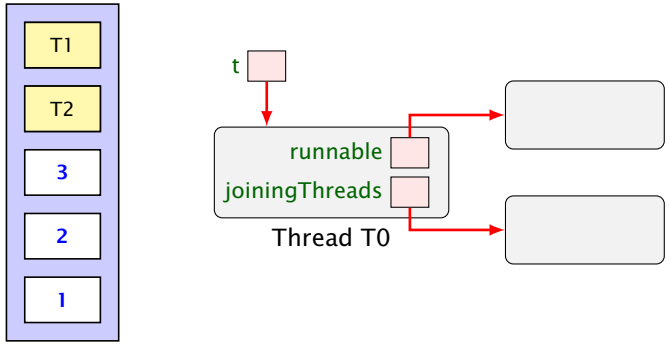
Die Join-Operation



Erläuterungen

- Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- Dabei gehen sie konzeptuell in einen Zustand joining über und werden aus der Menge der ausführbaren Threads entfernt.
- Beendet sich ein Thread, werden alle Threads, die auf ihn warteten, wieder aktiviert...

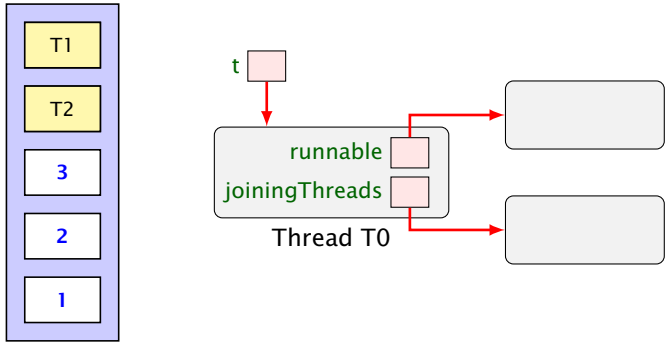
Die Join-Operation



Erläuterungen

- Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- Dabei gehen sie konzeptuell in einen Zustand joining über und werden aus der Menge der ausführbaren Threads entfernt.
- Beendet sich ein Thread, werden alle Threads, die auf ihn warteten, wieder aktiviert...

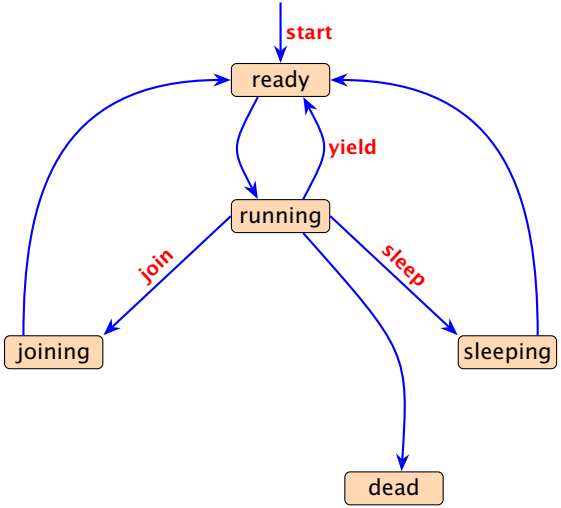
Die Join-Operation



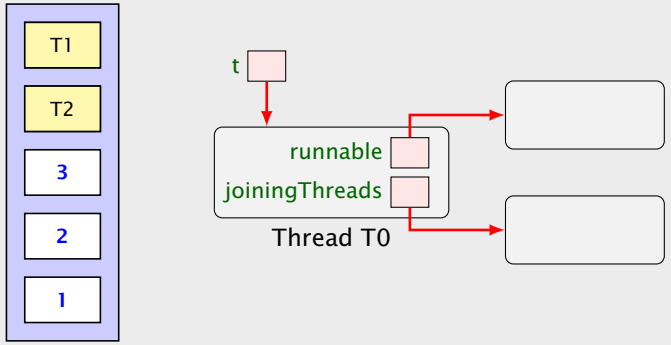
Erläuterungen

- Für jedes Threadobjekt `t` gibt es eine Schlange `ThreadQueue joiningThreads`.
- Threads, die auf Beendigung des Threads `t` warten, werden in diese Schlange eingefügt.
- Dabei gehen sie konzeptuell in einen Zustand joining über und werden aus der Menge der ausführbaren Threads entfernt.
- Beendet sich ein Thread, werden alle Threads, die auf ihn warteten, wieder aktiviert...

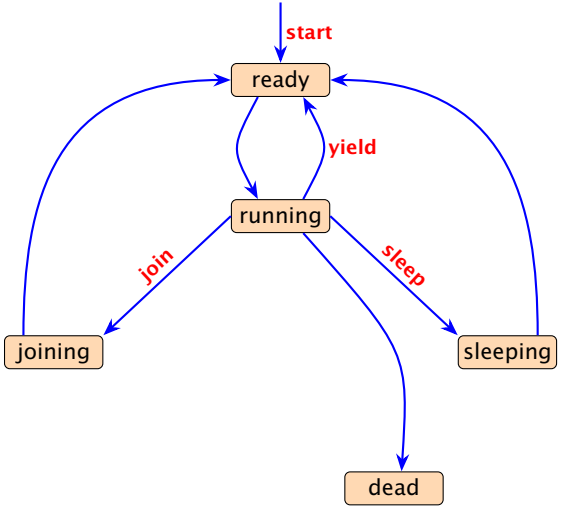
Threadzustände



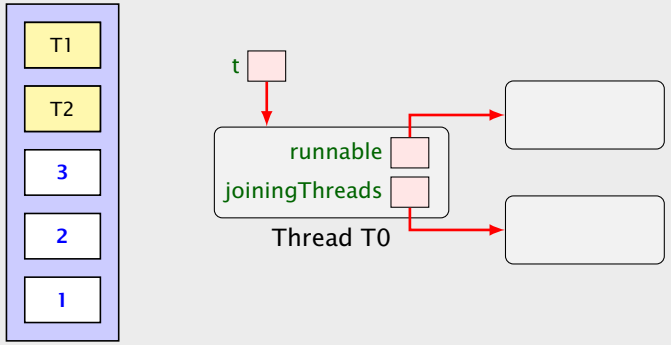
Die Join-Operation



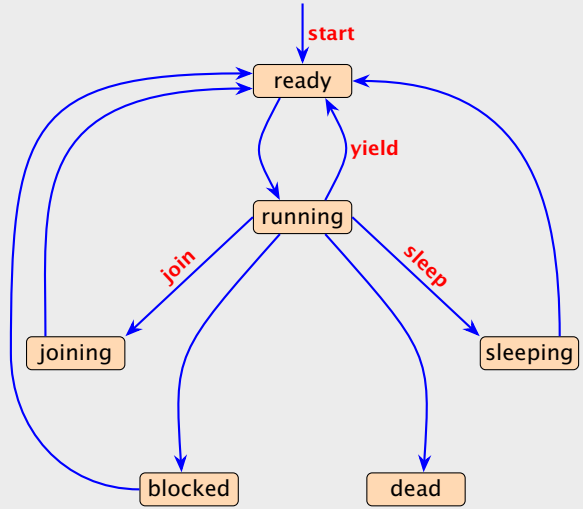
Threadzustände



Die Join-Operation



Threadzustände



Weiteres Beispiel

```
1  public class Join implements Runnable {
2      private static int count = 0;
3      private int n = count++;
4      private static Thread[] task = new Thread[3];
5      public void run() {
6          try {
7              if (n>0) {
8                  task[n-1].join();
9                  System.out.println("Thread-"+n+
10                         " joined Thread-"+(n-1));
11             }
12         } catch (InterruptedException e) {
13             System.err.println(e.toString());
14         }
15     } // continued...
```

Weiteres Beispiel

```
16      public static void main(String[] args) {
17          for(int i=0; i<3; i++)
18              task[i] = new Thread(new Join());
19          for(int i=0; i<3; i++)
20              task[i].start();
21      }
22  } // end of class Join
```

liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```


Weiteres Beispiel

```
1  public class Join implements Runnable {
2      private static int count = 0;
3      private int n = count++;
4      private static Thread[] task = new Thread[3];
5      public void run() {
6          try {
7              if (n>0) {
8                  task[n-1].join();
9                  System.out.println("Thread-"+n+
10                         " joined Thread-"+(n-1));
11             }
12         } catch (InterruptedException e) {
13             System.err.println(e.toString());
14         }
15     } // continued...
```

Variation

```
1  public class CW implements Runnable {
2      private static int count = 0;
3      private int n = count++;
4      private static Thread[] task = new Thread[3];
5      public void run() {
6          try { task[(n+1) % 3].join(); }
7          catch (InterruptedException e) {
8              System.err.println(e.toString());
9          }
10     }
11     public static void main(String[] args) {
12         for(int i=0; i<3; i++)
13             task[i] = new Thread(new CW());
14         for(int i=0; i<3; i++) task[i].start();
15     }
16 } // end of class CW
```


Weiteres Beispiel

```
16     public static void main(String[] args) {
17         for(int i=0; i<3; i++)
18             task[i] = new Thread(new Join());
19         for(int i=0; i<3; i++)
20             task[i].start();
21     }
22 } // end of class Join
```

liefert:

```
> java Join
Thread-1 joined Thread-0
Thread-2 joined Thread-1
```

Variation

```
1  public class CW implements Runnable {
2      private static int count = 0;
3      private int n = count++;
4      private static Thread[] task = new Thread[3];
5      public void run() {
6          try { task[(n+1) % 3].join(); }
7          catch (InterruptedException e) {
8              System.err.println(e.toString());
9          }
10     }
11     public static void main(String[] args) {
12         for(int i=0; i<3; i++)
13             task[i] = new Thread(new CW());
14         for(int i=0; i<3; i++) task[i].start();
15     }
16 } // end of class CW
```


Variation

- Das Programm terminiert möglicherweise nicht...
- `task[0]` wartet auf `task[1]`,
  `task[1]` wartet auf `task[2]`,
  `task[2]` wartet auf `task[0]`

  ist möglich...

Variation

- Das Programm terminiert möglicherweise nicht...
- `task[0]` wartet auf `task[1]`,
  `task[1]` wartet auf `task[2]`,
  `task[2]` wartet auf `task[0]`

  ist möglich...

Deadlock

- Jeder Thread geht in einen Wartezustand (hier: joining) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch Circular Wait oder Deadlock oder Verklemmung — eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte.

16.2 Monitore

- Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind Monitore.

Deadlock

- Jeder Thread geht in einen Wartezustand (hier: joining) über und wartet auf einen anderen Thread.
- Dieses Phänomen heißt auch Circular Wait oder Deadlock oder Verklemmung — eine unangenehme Situation, die man in seinen Programmen tunlichst vermeiden sollte.

Beispiel — Erhöhen einer Variablen

```
1  public class Inc implements Runnable {
2      private static int x = 0;
3      private static void pause(int t) {
4          try {
5              Thread.sleep((int) (Math.random()*t*1000));
6          } catch (InterruptedException e) {
7              System.err.println(e.toString());
8          }
9      }
10     public void run() {
11         String s = Thread.currentThread().getName();
12         pause(3); int y = x;
13         System.out.println(s+ " read "+y);
14         pause(4); x = y+1;
15         System.out.println(s+ " wrote "+(y+1));
16     }
17 // continued...
```

16.2 Monitore

- Damit Threads sinnvoll miteinander kooperieren können, müssen sie miteinander Daten austauschen.
- Zugriff mehrerer Threads auf eine gemeinsame Variable ist problematisch, weil nicht feststeht, in welcher Reihenfolge die Threads auf die Variable zugreifen.
- Ein Hilfsmittel, um geordnete Zugriffe zu garantieren, sind Monitore.

Beispiel

```
18      public static void main(String[] args) {
19          (new Thread(new Inc())).start();
20          pause(2);
21          (new Thread(new Inc())).start();
22          pause(2);
23          (new Thread(new Inc())).start();
24      }
25  } // end of class Inc
```

- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für 3 Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

Beispiel — Erhöhen einer Variablen

```
1  public class Inc implements Runnable {
2      private static int x = 0;
3      private static void pause(int t) {
4          try {
5              Thread.sleep((int) (Math.random()*t*1000));
6          } catch (InterruptedException e) {
7              System.err.println(e.toString());
8          }
9      }
10     public void run() {
11         String s = Thread.currentThread().getName();
12         pause(3); int y = x;
13         System.out.println(s+ " read "+y);
14         pause(4); x = y+1;
15         System.out.println(s+ " wrote "+(y+1));
16     }
17 // continued...
```

Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

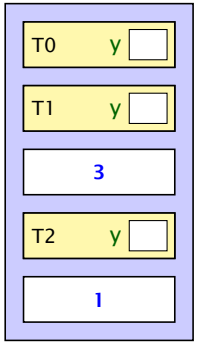
`x` wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Beispiel

```
18      public static void main(String[] args) {
19          (new Thread(new Inc())).start();
20          pause(2);
21          (new Thread(new Inc())).start();
22          pause(2);
23          (new Thread(new Inc())).start();
24      }
25  } // end of class Inc
```

- `public static Thread currentThread();` liefert (eine Referenz auf) das ausführende Thread-Objekt.
- `public final String getName();` liefert den Namen des Thread-Objekts.
- Das Programm legt für 3 Objekte der Klasse `Inc` Threads an.
- Die Methode `run()` inkrementiert die Klassen-Variable `x`.

Erklärung



x 0

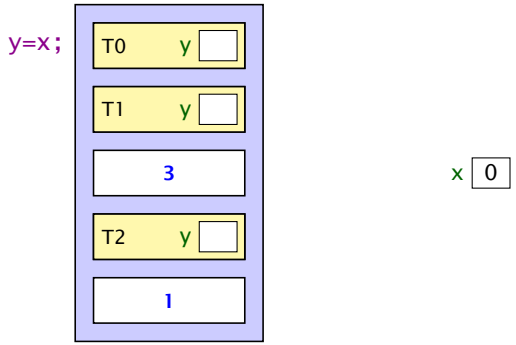
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



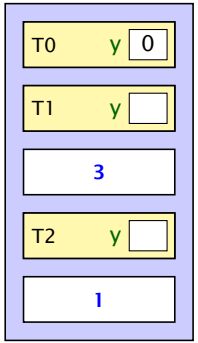
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



x 0

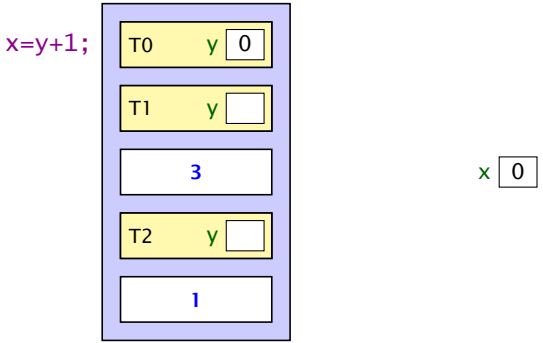
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



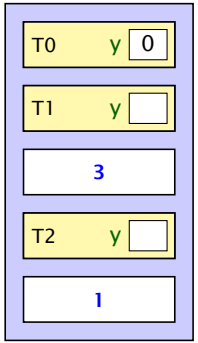
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



x 1

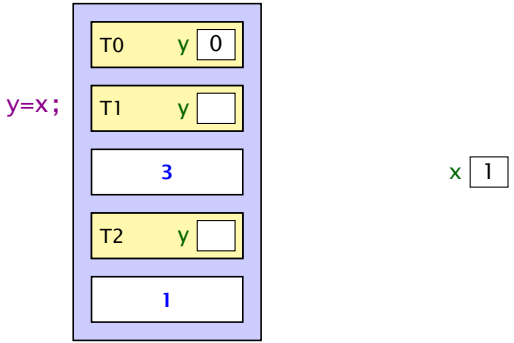
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



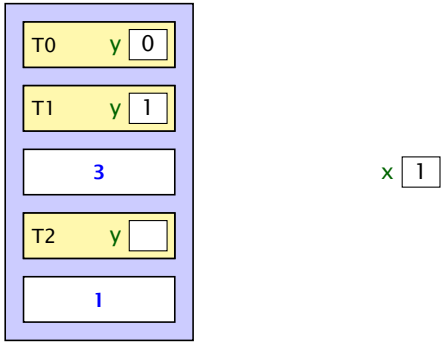
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



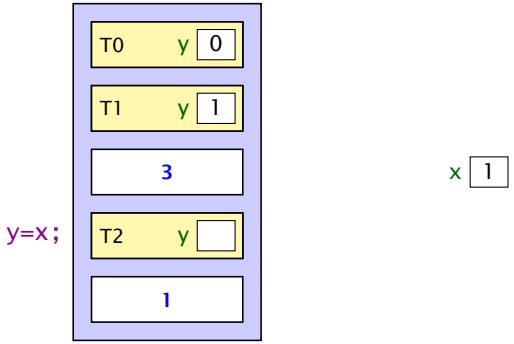
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



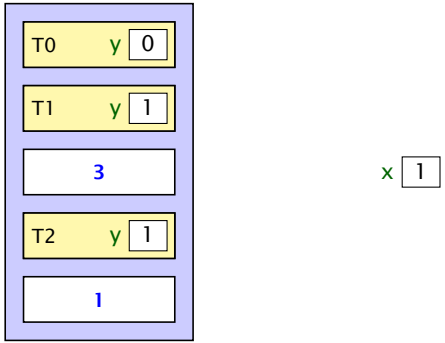
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



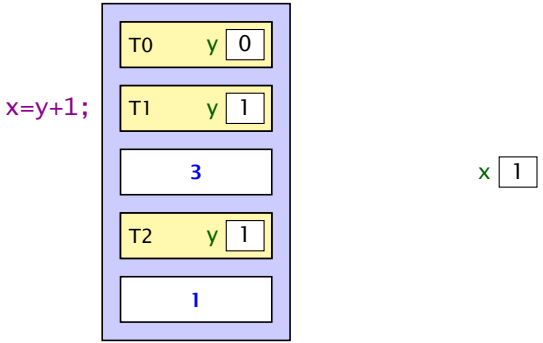
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



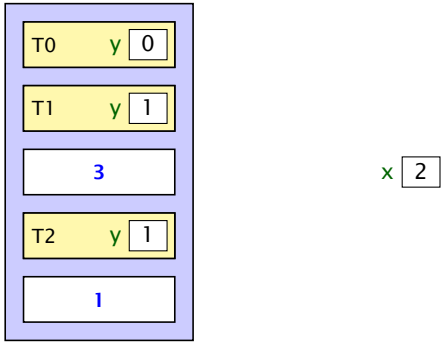
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



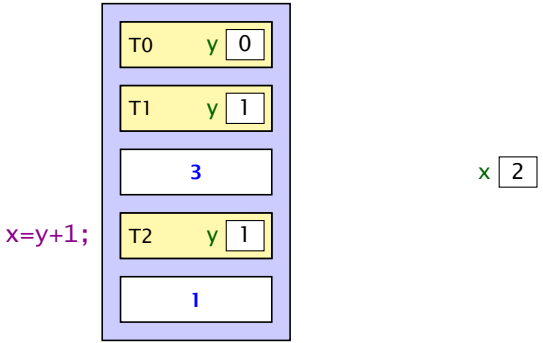
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



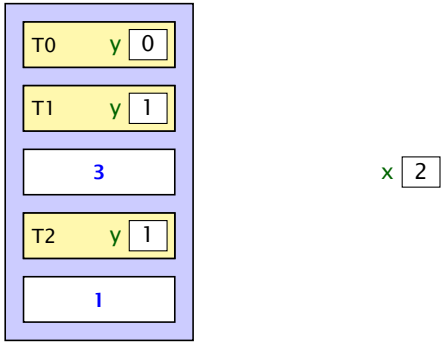
Beispiel

Mögliche Ausführung

```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

Erklärung



Beispiel

Mögliche Ausführung

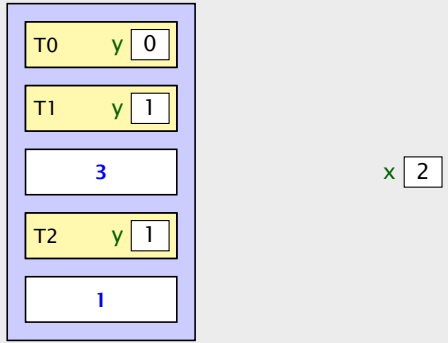
```
> java Inc
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-2 read 1
Thread-1 wrote 2
Thread-2 wrote 2
```

x wurde dreimal erhöht, hat aber am Ende den Wert 2!!!

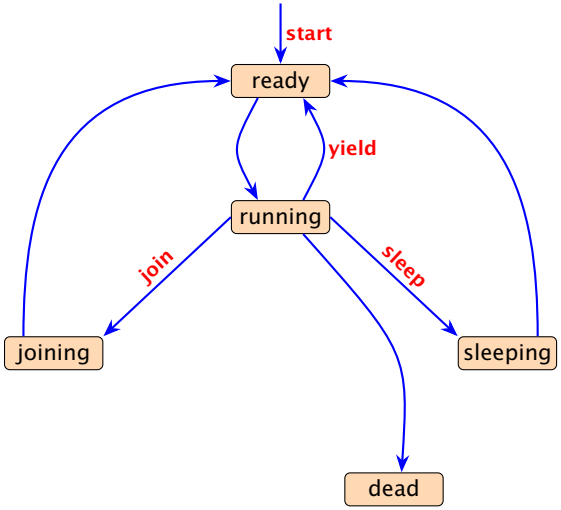
Monitore — Idee

- Inkrementieren der Variable x sollte ein atomarer Schritt sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- Mithilfe des Schlüsselworts `synchronized` kennzeichnen wir Objekt-Methoden einer Klasse L als ununterbrechbar.
- Für jedes Objekt `obj` der Klasse L kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer `synchronized`-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man kritischen Abschnitt („critical section“) für die gemeinsame Resource `obj`.
- Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt `obj` eintreten, werden alle bis auf einen blockiert.

Erklärung



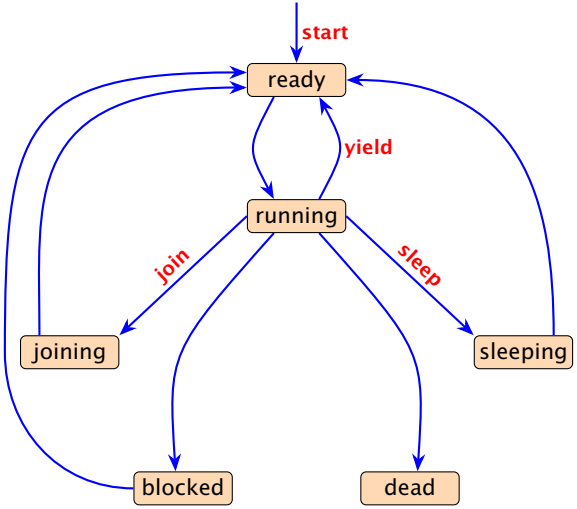
Threadzustände



Monitore — Idee

- Inkrementieren der Variable `x` sollte ein atomarer Schritt sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- Mithilfe des Schlüsselworts `synchronized` kennzeichnen wir Objekt-Methoden einer Klasse L als ununterbrechbar.
- Für jedes Objekt `obj` der Klasse L kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer `synchronized`-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man kritischen Abschnitt („critical section“) für die gemeinsame Resource `obj`.
- Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt `obj` eintreten, werden alle bis auf einen blockiert.

Threadzustände



Monitore — Idee

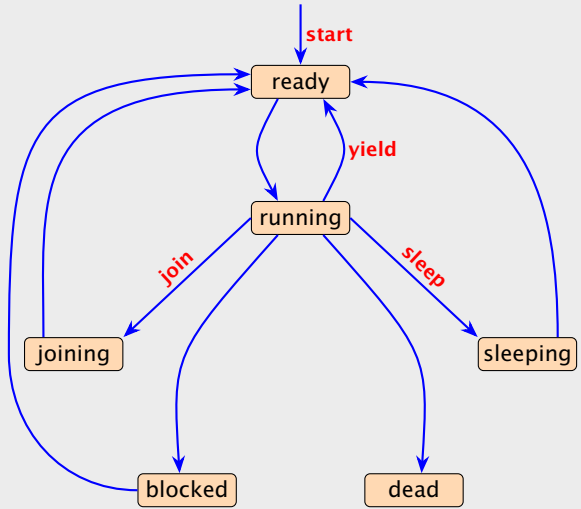
- Inkrementieren der Variable `x` sollte ein atomarer Schritt sein, d.h. nicht von parallel laufenden Threads unterbrochen werden können.
- Mithilfe des Schlüsselworts `synchronized` kennzeichnen wir Objekt-Methoden einer Klasse L als ununterbrechbar.
- Für jedes Objekt `obj` der Klasse L kann zu jedem Zeitpunkt nur ein Aufruf `obj.synchMeth(...)` einer `synchronized`-Methode `synchMeth()` ausgeführt werden. Die Ausführung einer solchen Methode nennt man kritischen Abschnitt („critical section“) für die gemeinsame Resource `obj`.
- Wollen mehrere Threads gleichzeitig in ihren kritischen Abschnitt für das Objekt `obj` eintreten, werden alle bis auf einen blockiert.

Locks

- Ein Objekt `obj` mit `synchronized`-Methoden verfügt über:
  1. boolesches Flag`boolean locked`; sowie
  2. eine Warteschlange `ThreadQueue blockedThreads`.
- Vor Betreten seines kritischen Abschnitts führt ein Thread (implizit) die atomare Operation `obj.lock()` aus:

```
private void lock() {
    if (!locked) locked = true; // betrete krit. Abschnitt
    else {                      // Lock bereits vergeben
        Thread t = Thread.currentThread();
        blockedThreads.enqueue(t);
        t.state = blocked;       // blockiere
    }
} // end of lock()
```

Threadzustände



Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

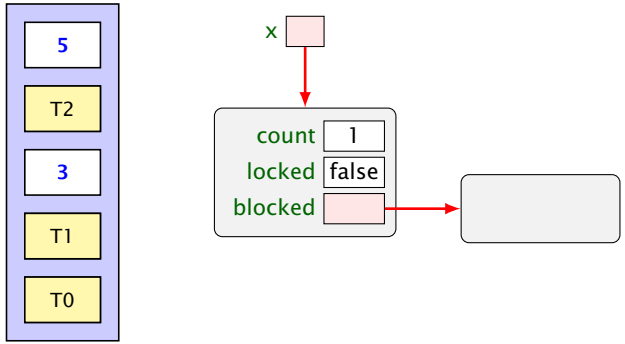
- Dieses Konzept nennt man Monitor.

Locks

- Ein Objekt obj mit synchronized-Methoden verfügt über:
  1. boolesches Flagboolean locked; sowie
  2. eine Warteschlange ThreadQueue blockedThreads.
- Vor Betreten seines kritischen Abschnitts führt ein Thread (implizit) die atomare Operation obj.lock() aus:

```
private void lock() {
    if (!locked) locked = true; // betrete krit. Abschnitt
    else {                      // Lock bereits vergeben
        Thread t = Thread.currentThread();
        blockedThreads.enqueue(t);
        t.state = blocked;       // blockiere
    }
} // end of lock()
```

Beispiel



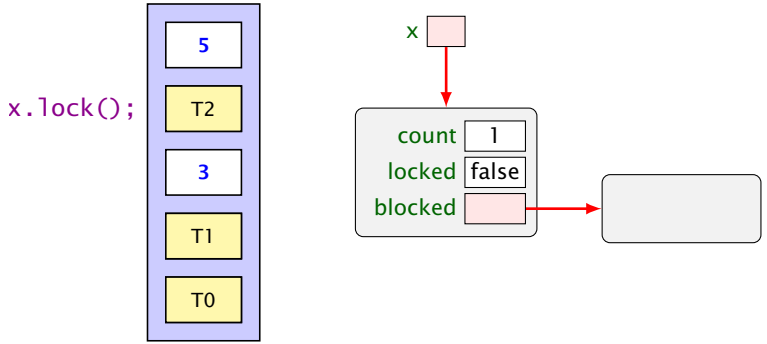
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



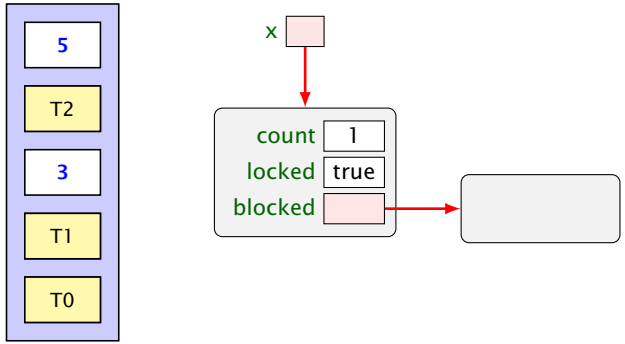
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



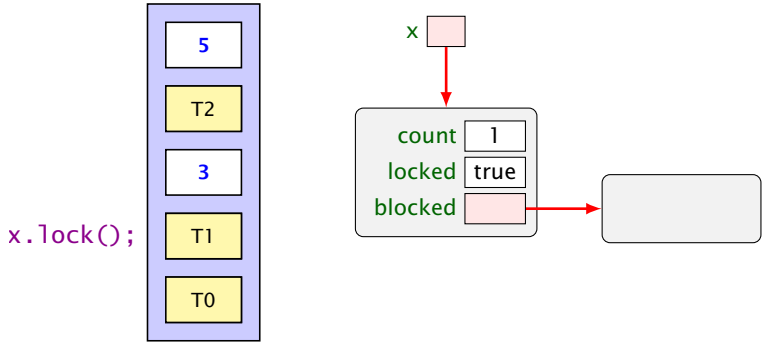
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



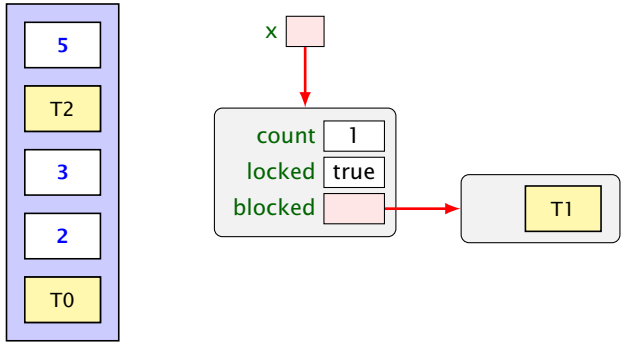
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



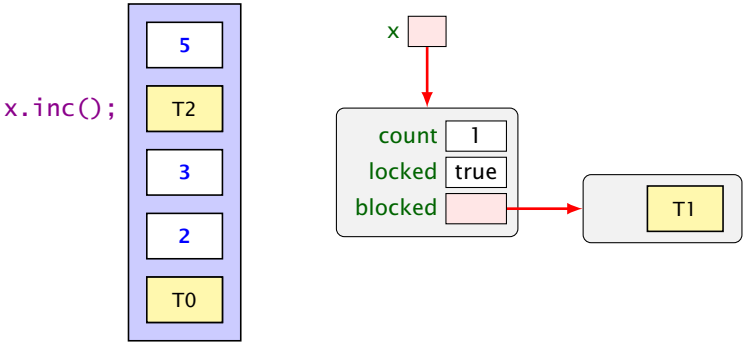
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



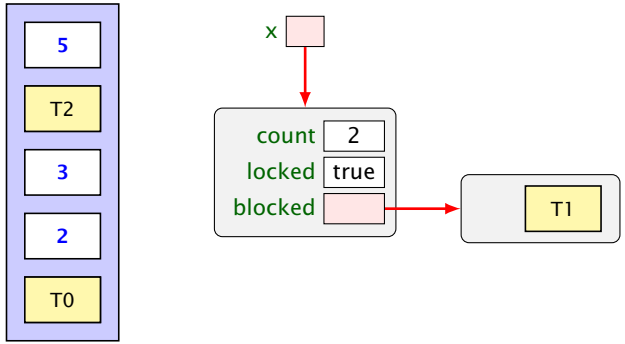
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



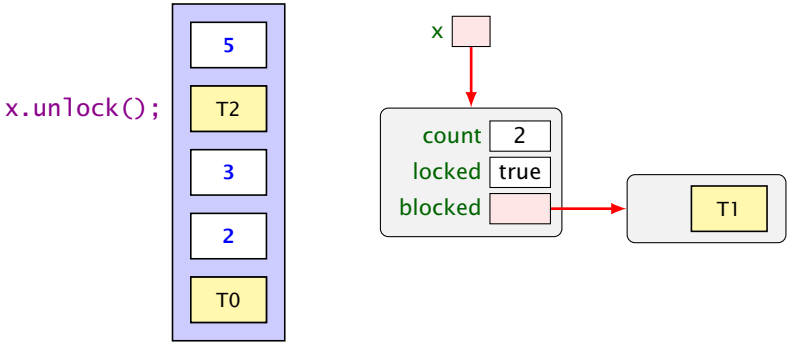
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



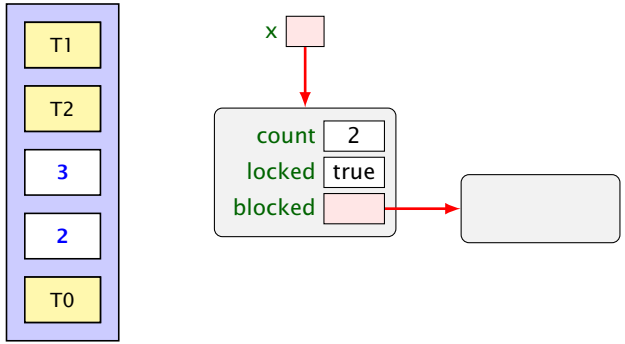
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



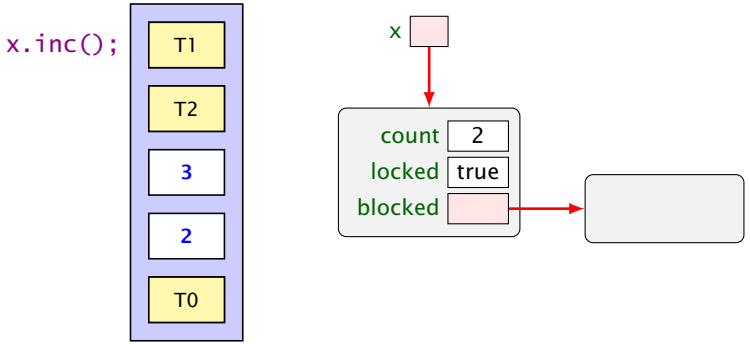
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



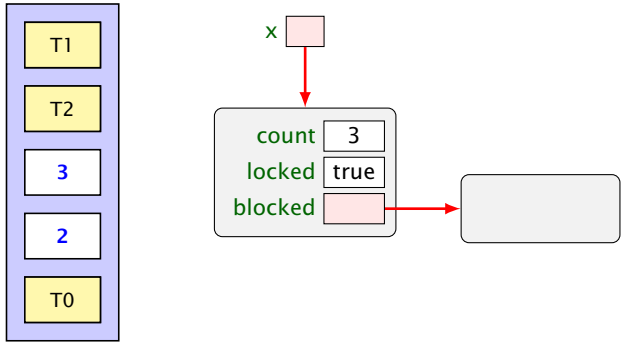
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



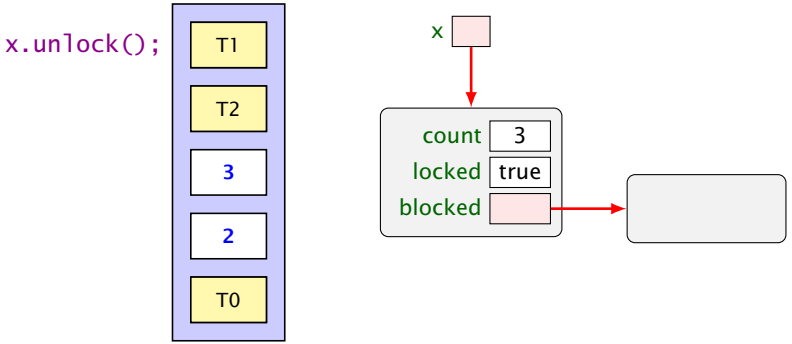
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



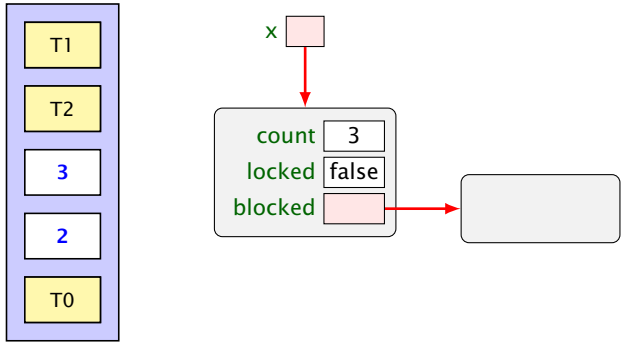
Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Beispiel



Locks

- Verlässt ein Thread seinen kritischen Abschnitt für obj (evt. auch mittels einer Exception), führt er (implizit) die atomare Operation obj.unlock() aus:

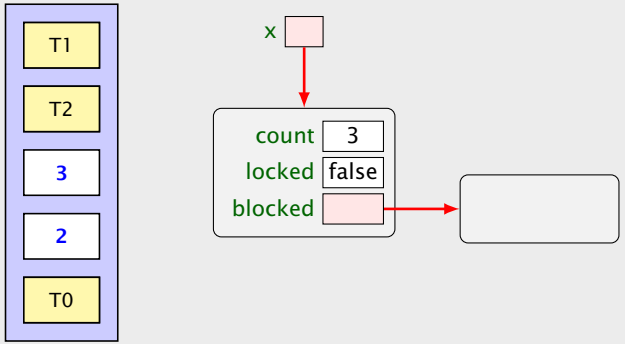
```
1 private void unlock() {
2     if (blockedThreads.empty())
3         locked = false; // Lock frei geben
4     else {              // Lock weiterreichen
5         Thread t = blockedThreads.dequeue();
6         t.state = ready;
7     }
8 } // end of unlock()
```

- Dieses Konzept nennt man Monitor.

Implementierung

```
class Count {
    private int count = 0;
    public synchronized void inc() {
        String s = Thread.currentThread().getName();
        int y=count; System.out.println(s+" read "+y);
        count=y+1;   System.out.println(s+" wrote "+count);
    }
} // end of class Count
public class IncSync implements Runnable {
    private static Count x = new Count();
    public void run() { x.inc(); }
    public static void main(String[] args) {
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
    }
} // end of class IncSync
```

Beispiel



Beispiel

liefert:

```
> java IncSync
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-1 wrote 2
Thread-2 read 2
Thread-2 wrote 3
```

Achtung:

- Die Operation `lock()` erfolgt nur, wenn der Thread nicht bereits vorher das Lock des Objekts erworben hat.
- Ein Thread, der das Lock eines Objekts `obj` besitzt, kann weitere Methoden für `obj` aufrufen, ohne sich selbst zu blockieren.

Implementierung

```
class Count {
    private int count = 0;
    public synchronized void inc() {
        String s = Thread.currentThread().getName();
        int y=count; System.out.println(s+" read "+y);
        count=y+1;   System.out.println(s+" wrote "+count);
    }
} // end of class Count
public class IncSync implements Runnable {
    private static Count x = new Count();
    public void run() { x.inc(); }
    public static void main(String[] args) {
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
        (new Thread(new IncSync())).start();
    }
} // end of class IncSync
```

Locks — im Detail

- Um das zu garantieren, legt ein Thread für jedes Objekt `obj`, dessen Lock er nicht besitzt, aber erwerben will, einen neuen Zähler an:

```
int countLock[obj] = 0;
```

- Bei jedem Aufruf einer `synchronized`-Methode `m(...)` für `obj` wird der Zähler inkrementiert, für jedes Verlassen (auch mittels Exceptions) dekrementiert:

```
if (0 == countLock[obj]++) lock();
obj.m(...)
if (--countLock[obj] == 0) unlock();
```

- `lock()` und `unlock()` werden nur ausgeführt, wenn

```
(countLock[obj] == 0)
```

Beispiel

liefert:

```
> java IncSync
Thread-0 read 0
Thread-0 wrote 1
Thread-1 read 1
Thread-1 wrote 2
Thread-2 read 2
Thread-2 wrote 3
```

Achtung:

- Die Operation `lock()` erfolgt nur, wenn der Thread nicht bereits vorher das Lock des Objekts erworben hat.
- Ein Thread, der das Lock eines Objekts `obj` besitzt, kann weitere Methoden für `obj` aufrufen, ohne sich selbst zu blockieren.

Beispiel: synchronized

```
1  public class StopThread extends Thread {
2      private static boolean stopRequested;
3
4      public void run() {
5          int i = 0;
6          while (!stopRequested)
7              i++;
8      }
9      public static void main(String[] args) {
10         Thread background = new StopThread();
11         background.start();
12         try {Thread.sleep(5);}
13         catch (InterruptedException e) {};
14         stopRequested = true;
15     }
16 }
```

Locks — im Detail

- Um das zu garantieren, legt ein Thread für jedes Objekt obj, dessen Lock er nicht besitzt, aber erwerben will, einen neuen Zähler an:

  `int countLock[obj] = 0;`

- Bei jedem Aufruf einer `synchronized`-Methode `m(...)` für obj wird der Zähler inkrementiert, für jedes Verlassen (auch mittels Exceptions) dekrementiert:

```
if (0 == countLock[obj]++) lock();
obj.m(...)
if (--countLock[obj] == 0) unlock();
```

- `lock()` und `unlock()` werden nur ausgeführt, wenn

  `(countLock[obj] == 0)`

Beispiel: synchronized

Das Programm terminiert (eventuell) nicht.

Die Änderung der Variablen `stopRequested` wird dem anderen Prozess nie mitgeteilt.

Wenn man auf die Variable durch `synchronized`-Methoden zugreift, ist sichergestellt, dass die Kommunikation korrekt durchgeführt wird.

Beispiel: synchronized

```
1  public class StopThread extends Thread {
2      private static boolean stopRequested;
3
4      public void run() {
5          int i = 0;
6          while (!stopRequested)
7              i++;
8      }
9      public static void main(String[] args) {
10         Thread background = new StopThread();
11         background.start();
12         try {Thread.sleep(5);}
13         catch (InterruptedException e) {};
14         stopRequested = true;
15     }
16 }
```

Beispiel: synchronized

```
1  public class StopThreadCorrect extends Thread {
2      private static boolean stopRequested;
3      private static synchronized void setStop() {
4          stopRequested = true;
5      }
6      private static synchronized boolean getStop() {
7          return stopRequested;
8      }
9      public void run() {
10         int i = 0;
11         while (!getStop()) i++;
12     }
13     public static void main(String[] args) {
14         Thread background = new StopThreadCorrect();
15         background.start();
16         try {Thread.sleep(5);}
17         catch (InterruptedException e) {};
18         setStop();
19     }
20 }
```

Beispiel: synchronized

Das Programm terminiert (eventuell) nicht.

Die Änderung der Variablen `stopRequested` wird dem anderen Prozess nie mitgeteilt.

Wenn man auf die Variable durch `synchronized`-Methoden zugreift, ist sichergestellt, dass die Kommunikation korrekt durchgeführt wird.

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse Data (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl N von Data-Objekten aufnehmen kann.

Beispiel: synchronized

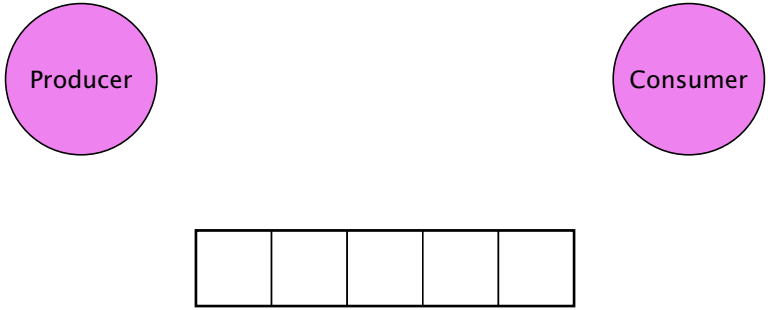
```
1  public class StopThreadCorrect extends Thread {
2      private static boolean stopRequested;
3      private static synchronized void setStop() {
4          stopRequested = true;
5      }
6      private static synchronized boolean getStop() {
7          return stopRequested;
8      }
9      public void run() {
10         int i = 0;
11         while (!getStop()) i++;
12     }
13     public static void main(String[] args) {
14         Thread background = new StopThreadCorrect();
15         background.start();
16         try {Thread.sleep(5);}
17         catch (InterruptedException e) {};
18         setStop();
19     }
20 }
```

16.3 Semaphore

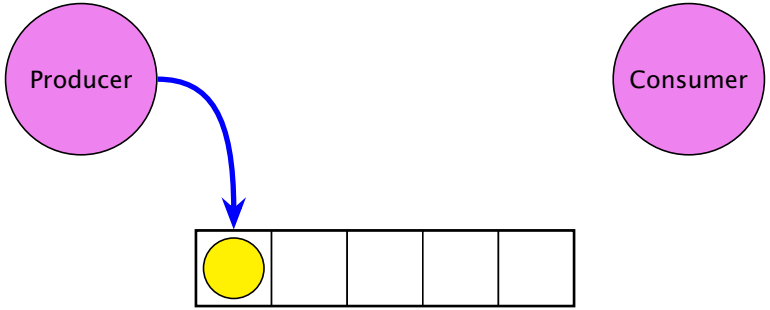
Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel



Beispiel



16.3 Semaphore

Aufgabe:

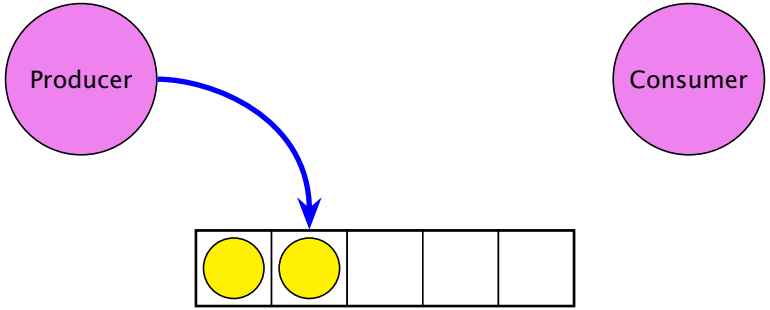
- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel

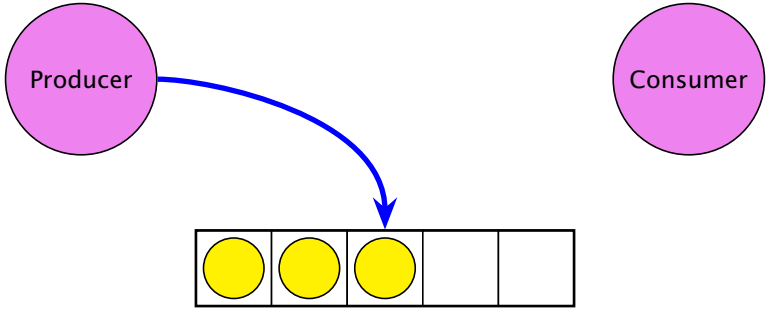


16.3 Semaphore

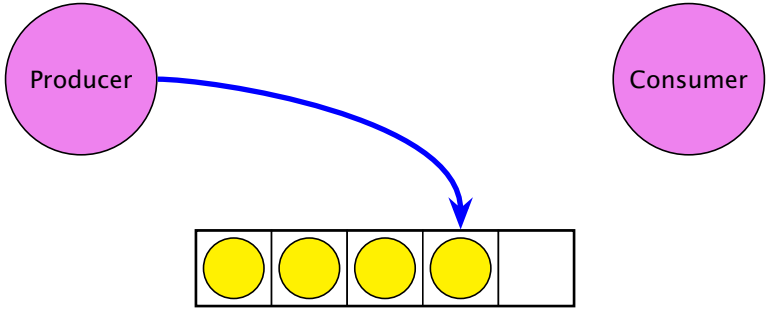
Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel



Beispiel

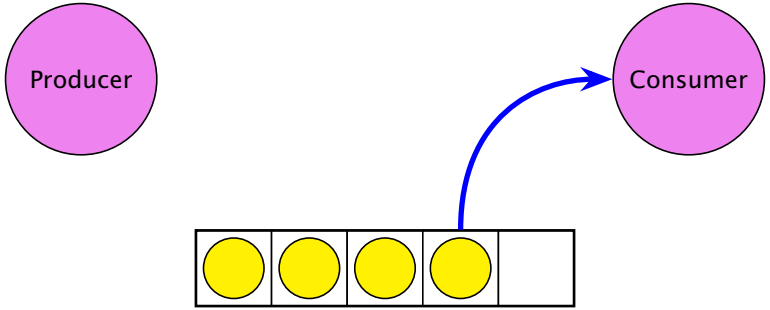


16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel

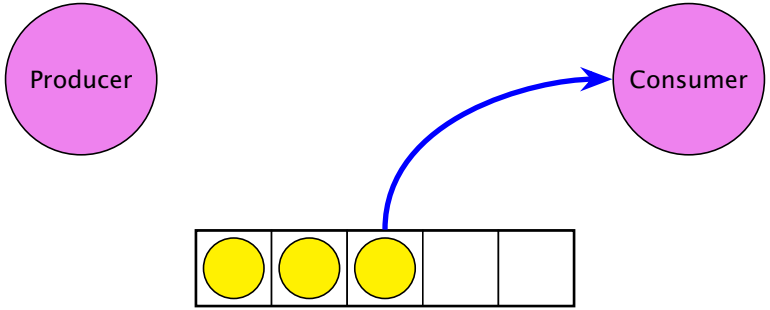


16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel

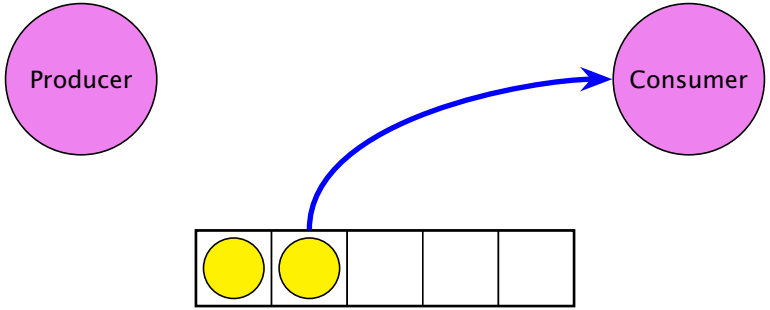


16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel



16.3 Semaphore

Aufgabe:

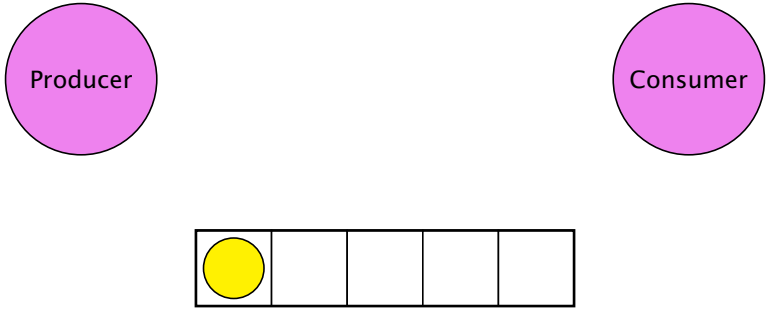
- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel

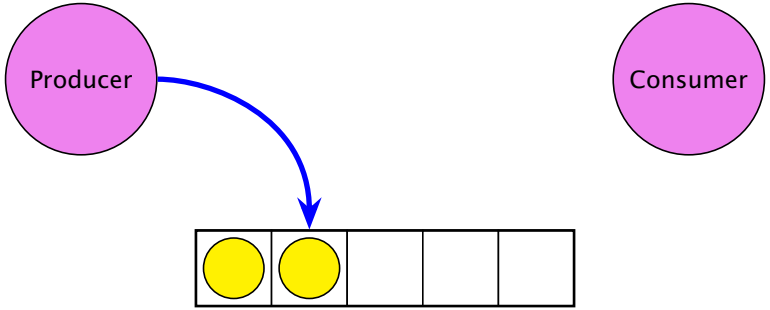


16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel

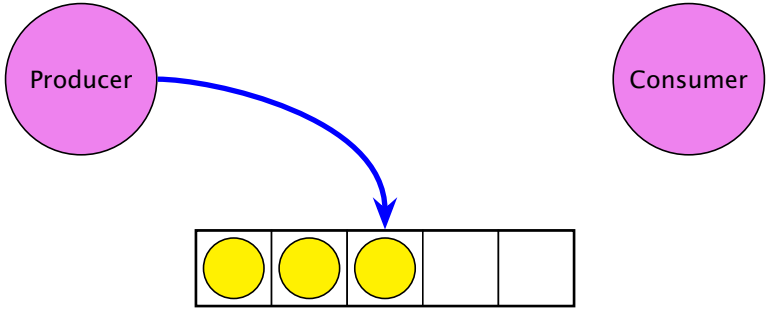


16.3 Semaphore

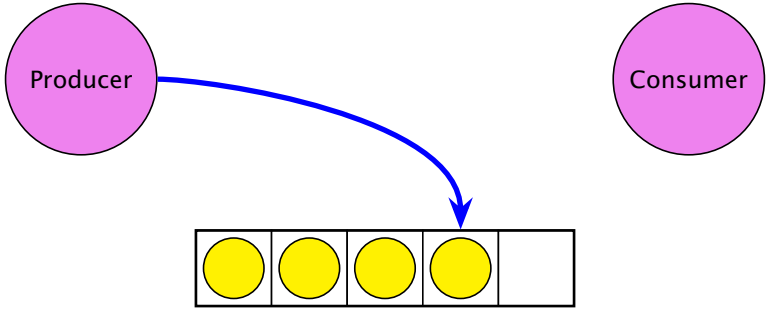
Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Beispiel



Beispiel



16.3 Semaphore

Aufgabe:

- Zwei Threads möchten mehrere/viele Daten-Objekte austauschen.
- Der eine Thread erzeugt die Objekte einer Klasse `Data` (Producer).
- Der andere konsumiert sie (Consumer).
- Zur übergabe dient ein Puffer, der eine feste Zahl `N` von `Data`-Objekten aufnehmen kann.

Consumer/Producer

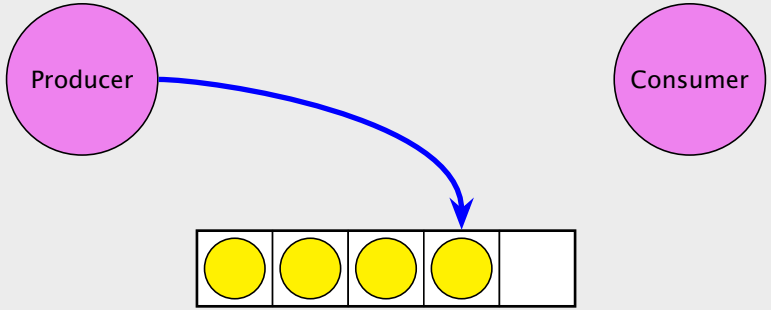
1.Idee

- Wir definieren eine Klasse `Buffer`, die (im wesentlichen) aus einem Feld der richtigen Größe, sowie zwei Verweisen `int first, last` zum Einfügen und Entfernen verfügt:

```
1 public class Buffer {
2     private int cap, free, first, last;
3     private Data[] a;
4     public Buffer(int n) {
5         free = cap = n; first = last = 0;
6         a = new Data[n];
7     }
8 // continued...
```

- Einfügen und Entnehmen sollen synchrone Operationen sein...

Beispiel



Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Consumer/Producer

1.Idee

- Wir definieren eine Klasse `Buffer`, die (im wesentlichen) aus einem Feld der richtigen Größe, sowie zwei Verweisen `int first, last` zum Einfügen und Entfernen verfügt:

```
1 public class Buffer {
2     private int cap, free, first, last;
3     private Data[] a;
4     public Buffer(int n) {
5         free = cap = n; first = last = 0;
6         a = new Data[n];
7     }
8 // continued...
```

- Einfügen und Entnehmen sollen synchrone Operationen sein...

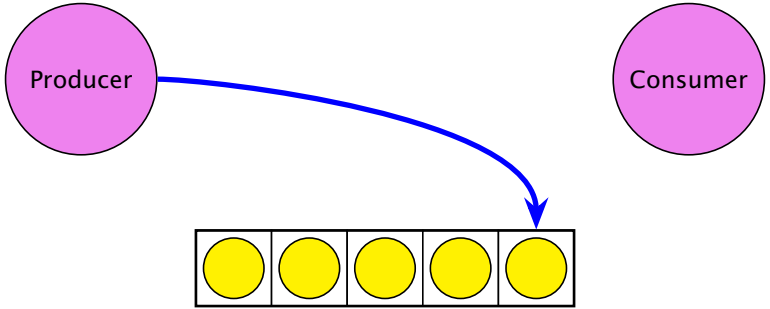
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



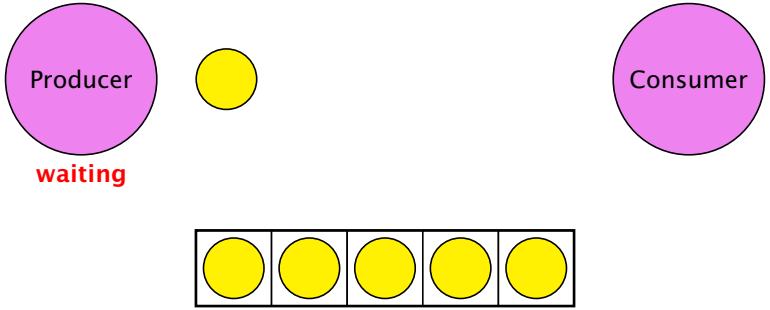
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



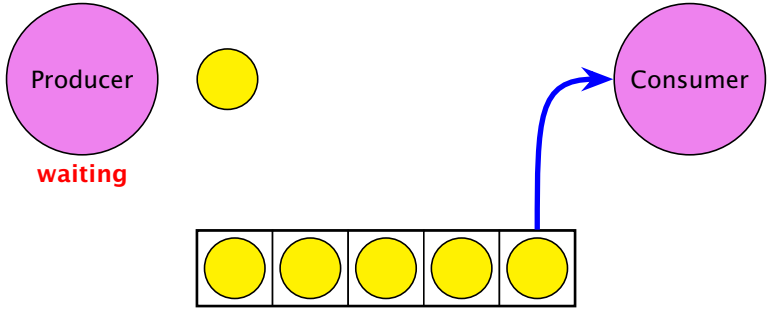
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



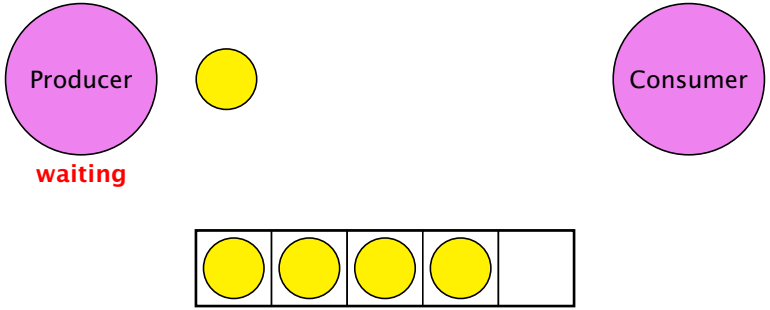
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



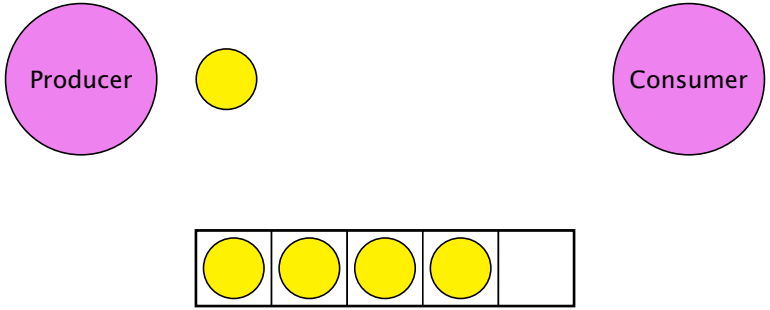
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



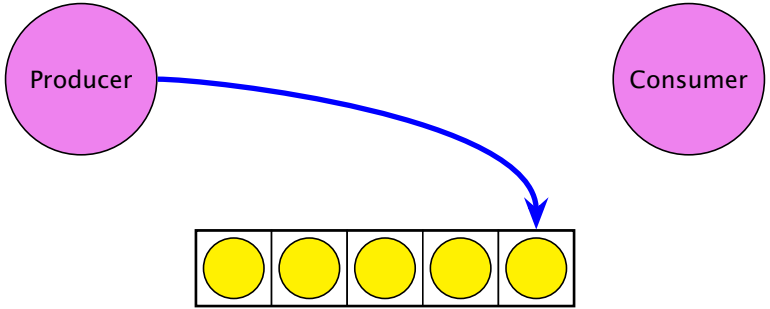
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



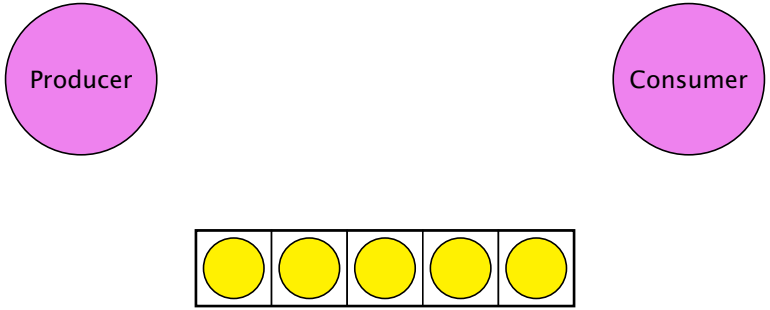
Consumer/Producer

Probleme

- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



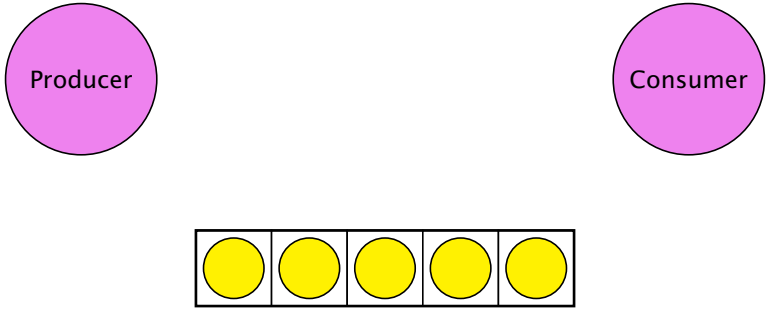
Consumer/Producer

Probleme

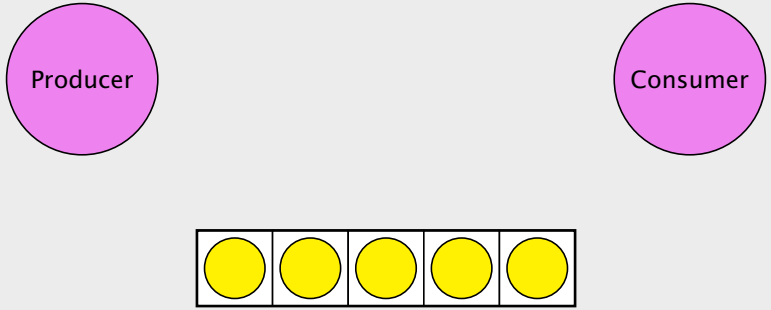
- Was macht der Consumer, wenn der Producer mit der Produktion nicht nachkommt, d.h. der Puffer leer ist?
- Was macht der Producer, wenn der Consumer mit der Weiterverarbeitung nicht nach kommt, d.h. der Puffer voll ist?

Lösungsvorschlag: Warten...

Beispiel



Beispiel



Umsetzung

- Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

```
public final void wait() throws InterruptedException;
public final void notify();
public final void notifyAll();
```

- Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt!!!

Umsetzung

- Jedes Objekt (mit `synchronized`-Methoden) verfügt über eine weitere Schlange `ThreadQueue waitingThreads` am Objekt wartender Threads sowie die Objekt-Methoden:

```
public final void wait() throws InterruptedException;
public final void notify();
public final void notifyAll();
```

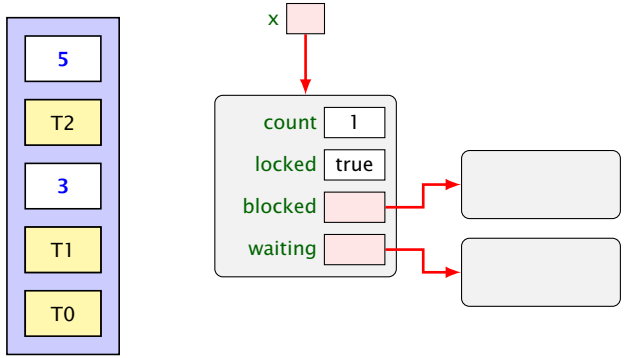
- Diese Methoden dürfen nur für Objekte aufgerufen werden, über deren Lock der Thread verfügt!!!

Umsetzung

- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
    Thread t = Thread.currentThread();
    t.state = waiting;
    waitingThreads.enqueue(t);
    unlock();
}
```

Beispiel

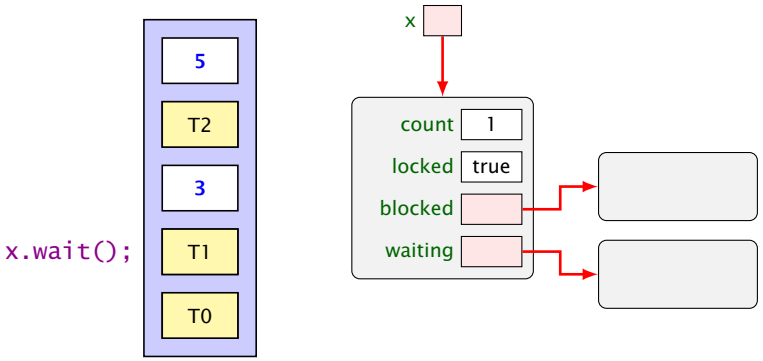


Umsetzung

- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
    Thread t = Thread.currentThread();
    t.state = waiting;
    waitingThreads.enqueue(t);
    unlock();
}
```

Beispiel

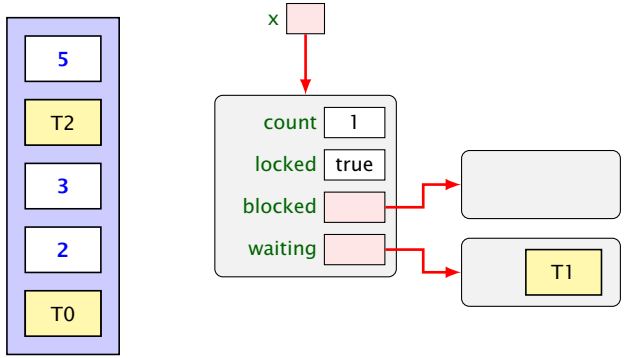


Umsetzung

- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
    Thread t = Thread.currentThread();
    t.state = waiting;
    waitingThreads.enqueue(t);
    unlock();
}
```

Beispiel

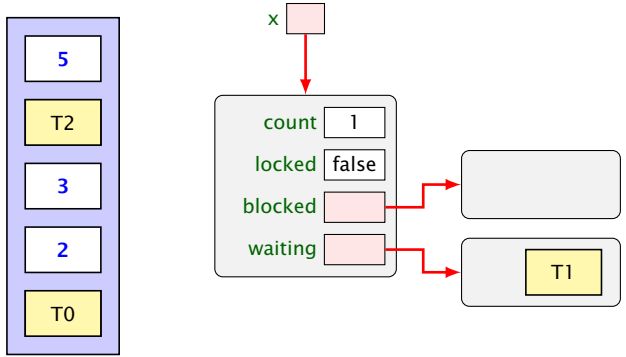


Umsetzung

- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
    Thread t = Thread.currentThread();
    t.state = waiting;
    waitingThreads.enqueue(t);
    unlock();
}
```

Beispiel



Umsetzung

- Ausführen von `wait();` setzt den Zustand des Threads auf waiting, reiht ihn in eine geeignete Warteschlange ein, und gibt das aktuelle Lock frei:

```
public void wait() throws InterruptedException {
    Thread t = Thread.currentThread();
    t.state = waiting;
    waitingThreads.enqueue(t);
    unlock();
}
```

Umsetzung

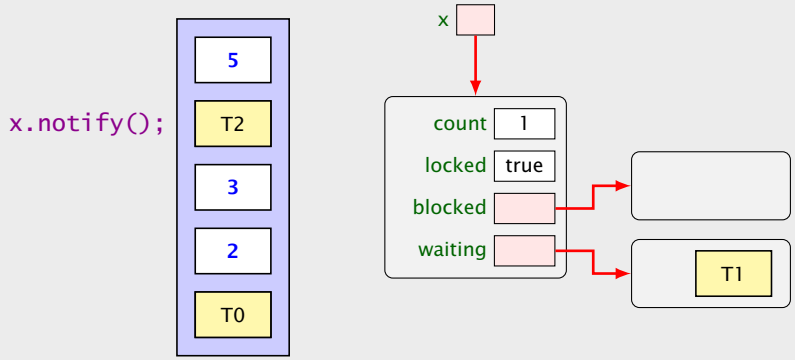
- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand blocked und fügt ihn in `blockedThreads` ein:

```
public void notify() {
    if (!waitingThreads.isEmpty()) {
        Thread t = waitingThreads.dequeue();
        t.state = blocked;
        blockedThreads.enqueue(t);
    }
}
```

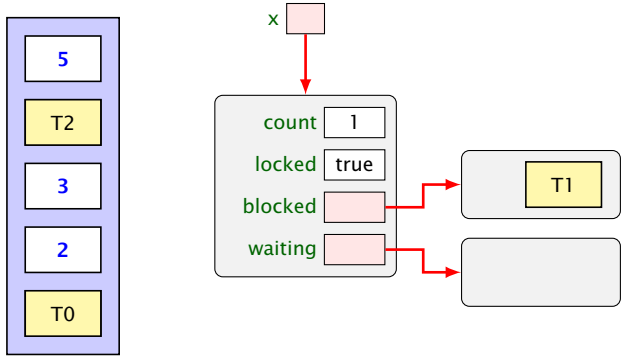
- `notifyAll();` weckt alle wartenden Threads auf d.h. fügt alle in `blockedThreads` ein:

```
public void notifyAll() {
    while (!waitingThreads.isEmpty()) notify();
}
```

Beispiel



Beispiel



Umsetzung

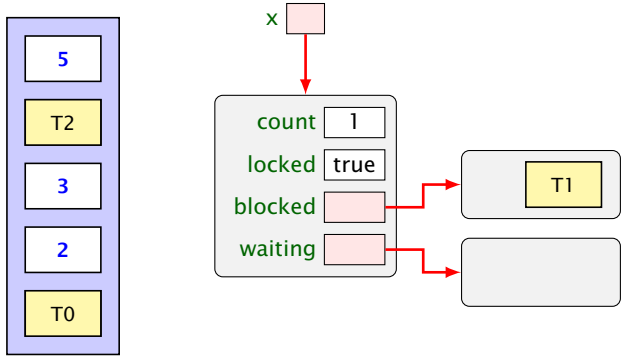
- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand blocked und fügt ihn in `blockedThreads` ein:

```
public void notify() {
    if (!waitingThreads.isEmpty()) {
        Thread t = waitingThreads.dequeue();
        t.state = blocked;
        blockedThreads.enqueue(t);
    }
}
```

- `notifyAll();` weckt alle wartenden Threads auf d.h. fügt alle in `blockedThreads` ein:

```
public void notifyAll() {
    while (!waitingThreads.isEmpty()) notify();
}
```

Beispiel



Umsetzung

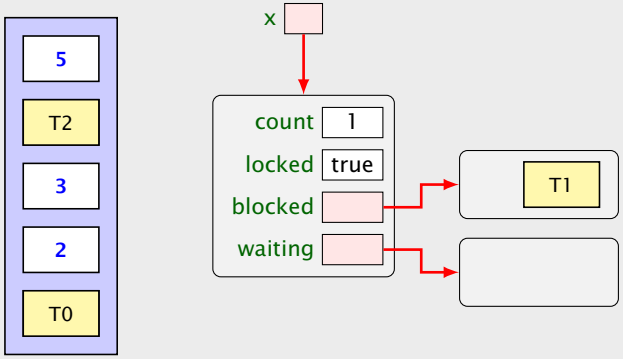
- Ausführen von `notify();` weckt den ersten Thread in der Warteschlange auf, d.h. versetzt ihn in den Zustand blocked und fügt ihn in `blockedThreads` ein:

```
public void notify() {
    if (!waitingThreads.isEmpty()) {
        Thread t = waitingThreads.dequeue();
        t.state = blocked;
        blockedThreads.enqueue(t);
    }
}
```

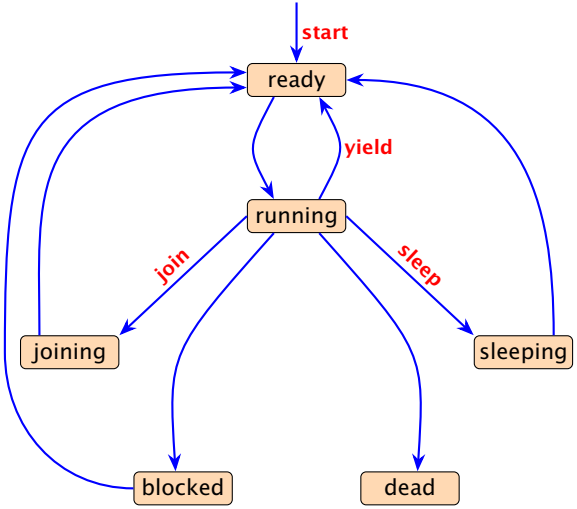
- `notifyAll();` weckt alle wartenden Threads auf d.h. fügt alle in `blockedThreads` ein:

```
public void notifyAll() {
    while (!waitingThreads.isEmpty()) notify();
}
```

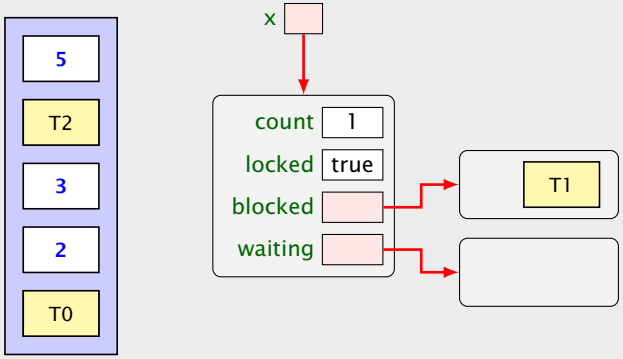
Beispiel



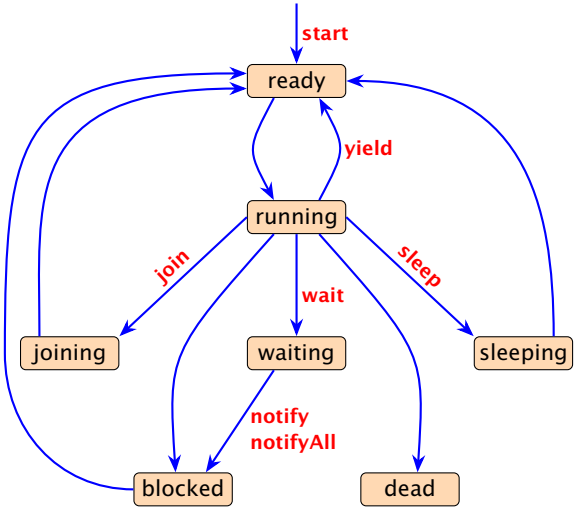
Threadzustände



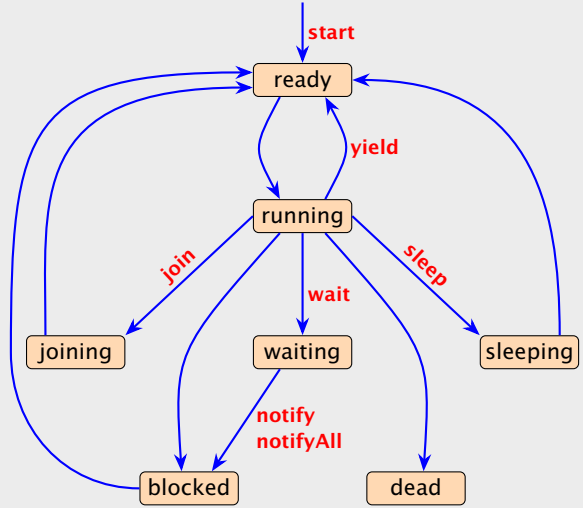
Beispiel



Threadzustände



Threadzustände



Implementierung

```
 9     public synchronized void produce(Data d)
10                           throws InterruptedException {
11         if (free==0) wait(); free--;
12         a[last] = d;
13         last = (last+1) % cap;
14         notify();
15     }
16     public synchronized Data consume()
17                           throws InterruptedException {
18         if (free==cap) wait(); free++;
19         Data result = a[first];
20         first = (first+1) % cap;
21         notify();
22         return result;
23     }
24 } // end of class Buffer2
```

Implementierung

```
 9     public synchronized void produce(Data d)
10                         throws InterruptedException {
11         if (free==0) wait(); free--;
12         a[last] = d;
13         last = (last+1) % cap;
14         notify();
15     }
16     public synchronized Data consume()
17                         throws InterruptedException {
18         if (free==cap) wait(); free++;
19         Data result = a[first];
20         first = (first+1) % cap;
21         notify();
22         return result;
23     }
24 } // end of class Buffer2
```


Erläuterungen

- Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers stets den Producer (und umgekehrt).
- Was aber, wenn es mehrere Producers gibt? Oder mehrere Consumers?

Erläuterungen

- Ist der Puffer voll, d.h. keine Zelle frei, legt sich der Producer schlafen.
- Ist der Puffer leer, d.h. alle Zellen frei, legt sich der Consumer schlafen.
- Gibt es für einen Puffer genau einen Producer und einen Consumer, weckt das `notify()` des Consumers stets den Producer (und umgekehrt).
- Was aber, wenn es mehrere Producers gibt? Oder mehrere Consumers?

Consumer/Producer

2. Idee: Wiederholung des Tests

- Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- Wecke nicht einen, sondern alle wartenden Threads auf...

```
 9    public synchronized void produce(Data d)
10                  throws InterruptedException {
11        while (free==0) wait();
12        free--;
13        a[last] = d;
14        last = (last+1) % cap;
15        notifyAll();
16    }
```

Consumer/Producer

```
16     public synchronized Data consume()
17                     throws InterruptedException {
18         while (free==cap) wait();
19         free++;
20         Data result = a[first];
21         first = (first+1) % cap;
22         notifyAll();
23         return result;
24     }
25 } // end of class Buffer2
```

- Wenn ein Platz im Puffer frei wird, werden sämtliche Threads aufgeweckt — obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann.

Consumer/Producer

2. Idee: Wiederholung des Tests

- Teste nach dem Aufwecken erneut, ob Zellen frei sind.
- Wecke nicht einen, sondern alle wartenden Threads auf...

```
 9     public synchronized void produce(Data d)
10                     throws InterruptedException {
11         while (free==0) wait();
12         free--;
13         a[last] = d;
14         last = (last+1) % cap;
15         notifyAll();
16     }
```

Consumer/Producer

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

```
1  public class Sema {
2      private int x;
3      public Sema(int n) { x = n; }
4      public synchronized void up() {
5          x++; if (x <= 0) this.notify();
6      }
7      public synchronized void down()
8                  throws InterruptedException {
9          x--; while (x < 0) this.wait();
10     }
11 } // end of class Sema
```

Consumer/Producer

```
16     public synchronized Data consume()
17                 throws InterruptedException {
18         while (free==cap) wait();
19         free++;
20         Data result = a[first];
21         first = (first+1) % cap;
22         notifyAll();
23         return result;
24     }
25 } // end of class Buffer2
```

- Wenn ein Platz im Puffer frei wird, werden sämtliche Threads aufgeweckt — obwohl evt. nur einer der Producer bzw. nur einer der Consumer aktiv werden kann.

Semaphore

- Ein Semaphor enthält eine private `int`-Objekt-Variable und bietet die `synchronized`-Methoden `up()` und `down()` an.
- `up()` erhöht die Variable, `down()` erniedrigt sie.
- Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Resourcen an.
  Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- Eine `up()`-Operation weckt genau einen wartenden Thread auf.

Consumer/Producer

3. Idee: Semaphore

- Producers und Consumers warten in verschiedenen Schlangen.
- Die Producers warten darauf, dass `free > 0` ist.
- Die Consumers warten darauf, dass `cap-free > 0` ist.

```
1  public class Sema {
2      private int x;
3      public Sema(int n) { x = n; }
4      public synchronized void up() {
5          x++; if (x <= 0) this.notify();
6      }
7      public synchronized void down()
8                  throws InterruptedException {
9          x--; while (x < 0) this.wait();
10     }
11 } // end of class Sema
```

Semaphore

- Ein Semaphor enthält eine private `int`-Objekt-Variable und bietet die `synchronized`-Methoden `up()` und `down()` an.
- `up()` erhöht die Variable, `down()` erniedrigt sie.
- Ist die Variable positiv, gibt sie die Anzahl der verfügbaren Resourcen an.
  Ist sie negativ, zählt sie die Anzahl der wartenden Threads.
- Eine `up()`-Operation weckt genau einen wartenden Thread auf.

Anwendung – 1. Versuch

```
1  public class BufferFaulty {
2      private int cap, first, last;
3      private Sema free, occupied;
4      private Data[] a;
5      public BufferFaulty(int n) {
6          cap = n;
7          first = last = 0;
8          a = new Data[n];
9          free = new Sema(n);
10         occupied = new Sema(0);
11     }
12 // continued...
```

Anwendung – 1. Versuch

```
12      public synchronized void produce(Data d)
13                              throws InterruptedException {
14          free.down();
15          a[last] = d;
16          last = (last+1) % cap;
17          occupied.up();
18      }
19      public synchronized Data consume()
20                              throws InterruptedException {
21          occupied.down();
22          Data result = a[first];
23          first = (first+1) % cap;
24          free.up();
25          return result;
26      }
27  }
```


Anwendung – 1. Versuch

```
1  public class BufferFaulty {
2      private int cap, first, last;
3      private Sema free, occupied;
4      private Data[] a;
5      public BufferFaulty(int n) {
6          cap = n;
7          first = last = 0;
8          a = new Data[n];
9          free = new Sema(n);
10         occupied = new Sema(0);
11     }
12 // continued...
```

Anwendung – 1. Versuch

```
12      public synchronized void produce(Data d)
13                              throws InterruptedException {
14          free.down();
15          a[last] = d;
16          last = (last+1) % cap;
17          occupied.up();
18      }
19      public synchronized Data consume()
20                              throws InterruptedException {
21          occupied.down();
22          Data result = a[first];
23          first = (first+1) % cap;
24          free.up();
25          return result;
26      }
27  }
```


Deadlock

- Gut gemeint — aber leider fehlerhaft...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück... nicht aber dasjenige für den Puffer!!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

2. Versuch – Entkopplung der Locks

```
12      // Methoden sind nicht synchronized
13      public void produce(Data d) throws
14                                  InterruptedException {
15          free.down();
16          synchronized (this) {
17              a[last] = d; last = (last+1) % cap;
18          }
19          occupied.up();
20      }
21      public Data consume() throws
22                              InterruptedException {
23          Data result; occupied.down();
24          synchronized (this) {
25              result = a[first]; first = (first+1) % cap;
26          }
27          free.up(); return result;
28      }
29  } // end of corrected class Buffer
```

Deadlock

- Gut gemeint — aber leider fehlerhaft...
- Jeder Producer benötigt zwei Locks gleichzeitig, um zu produzieren:
  1. dasjenige für den Puffer;
  2. dasjenige für einen Semaphor.
- Muss er für den Semaphor ein `wait()` ausführen, gibt er das Lock für den Semaphor wieder zurück... nicht aber dasjenige für den Puffer!!!
- Die Folge ist, dass niemand mehr eine Puffer-Operation ausführen kann, insbesondere auch kein `up()` mehr für den Semaphor $\Longrightarrow$ Deadlock

Erläuterung

- Das Statement `synchronized (obj ) { stmts }` definiert einen kritischen Bereich für das Objekt `obj`, in dem die Statement-Folge `stmts` ausgeführt werden soll.
- Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock.

2. Versuch – Entkopplung der Locks

```
12      // Methoden sind nicht synchronized
13      public void produce(Data d) throws
14                                  InterruptedException {
15          free.down();
16          synchronized (this) {
17              a[last] = d; last = (last+1) % cap;
18          }
19          occupied.up();
20      }
21      public Data consume() throws
22                              InterruptedException {
23          Data result; occupied.down();
24          synchronized (this) {
25              result = a[first]; first = (first+1) % cap;
26          }
27          free.up(); return result;
28      }
29  } // end of corrected class Buffer
```

Ziel:

- eine Datenstruktur soll gemeinsam von mehreren Threads benutzt werden.
- Jeder Thread soll (gefühlt) atomar auf die Datenstruktur zugreifen.
- Lesende Zugriffe sollen die Datenstruktur nicht verändern, schreibende Zugriffe dagegen können die Datenstruktur modifizieren.

Erläuterung

- Das Statement `synchronized (obj ) { stmts }` definiert einen kritischen Bereich für das Objekt `obj`, in dem die Statement-Folge `stmts` ausgeführt werden soll.
- Threads, die die neuen Objekt-Methoden `void produce(Data d)` bzw. `Data consume()` ausführen, benötigen zu jedem Zeitpunkt nur genau ein Lock.

16.4 RW-Locks

Ziel:

- eine Datenstruktur soll gemeinsam von mehreren Threads benutzt werden.
- Jeder Thread soll (gefühlt) atomar auf die Datenstruktur zugreifen.
- Lesende Zugriffe sollen die Datenstruktur nicht verändern, schreibende Zugriffe dagegen können die Datenstruktur modifizieren.

Implementierung

1. Idee: Synchronisiere Methodenaufrufe

```
1  public class HashTable<K,V> {
2      private List2<K,V> [] a;
3      private int n;
4      public HashTable (int n) {
5          a = new List2[n];
6          this.n = n;
7      }
8      public synchronized V lookup (K k) {...}
9      public synchronized void update (K k, V v) {...}
10 }
```

Implementierung

1. Idee: Synchronisiere Methodenaufrufe

```
1  public class HashTable<K,V> {
2      private List2<K,V> [] a;
3      private int n;
4      public HashTable (int n) {
5          a = new List2[n];
6          this.n = n;
7      }
8      public synchronized V lookup (K k) {...}
9      public synchronized void update (K k, V v) {...}
10 }
```


Arrays und Generics

Wir sagen `a = new List2[n]` anstatt `a = new List2<K,V>[n]`. Letzteres führt zu einem Compilerfehler.

```
1 // nicht moeglich...
2 List<Dog>[] a = new List<Dog>[100];
3 Object[] b = a;
4 List<Cat> cats = new List<Cat>();
5 cats.insert(new Cat("Garfield"));
6 // keine ArrayStoreException wegen Type-Erasure
7 b[0] = cats;
8 // das sollte eigentlich ok sein...
9 Dog d = a[0].remove(0);
```

Man sollte (wenn möglich) Generics und Arrays in Java nicht miteinander mischen.

Arrays und Generics

Wir sagen `a = new List2[n]` anstatt `a = new List2<K,V>[n]`. Letzteres führt zu einem Compilerfehler.

```
1 // nicht moeglich...
2 List<Dog>[] a = new List<Dog>[100];
3 Object[] b = a;
4 List<Cat> cats = new List<Cat>();
5 cats.insert(new Cat("Garfield"));
6 // keine ArrayStoreException wegen Type-Erasure
7 b[0] = cats;
8 // das sollte eigentlich ok sein...
9 Dog d = a[0].remove(0);
```

Man sollte (wenn möglich) Generics und Arrays in Java nicht miteinander mischen.

Diskussion

- Zu jedem Zeitpunkt darf nur ein Thread auf die HashTable zugreifen.
- Für schreibende Threads ist das evt. sinnvoll.
- Threads, die nur lesen, stören sich gegenseitig aber überhaupt nicht!

$\Longrightarrow$ **ReaderWriterLock**

Diskussion

- Zu jedem Zeitpunkt darf nur ein Thread auf die HashTable zugreifen.
- Für schreibende Threads ist das evt. sinnvoll.
- Threads, die nur lesen, stören sich gegenseitig aber überhaupt nicht!

$\Longrightarrow$ **ReaderWriterLock**

RW-Lock

- ist entweder im Lese-Modus, im Schreibmodus oder frei.
- Im Lese-Modus dürfen beliebig viele Leser eintreten, während sämtliche Schreiber warten müssen.
- Haben keine Leser mehr Interesse, ist das Lock wieder frei.
- Ist das Lock frei, darf ein Schreiber eintreten. Das RW-Lock wechselt nun in den Schreib-Modus.
- Im Schreib-Modus müssen sowohl Leser als auch weitere Schreiber warten.
- Ist ein Schreiber fertig, wird das Lock wieder frei gegeben...

RW-Lock

- ist entweder im Lese-Modus, im Schreibmodus oder frei.
- Im Lese-Modus dürfen beliebig viele Leser eintreten, während sämtliche Schreiber warten müssen.
- Haben keine Leser mehr Interesse, ist das Lock wieder frei.
- Ist das Lock frei, darf ein Schreiber eintreten. Das RW-Lock wechselt nun in den Schreib-Modus.
- Im Schreib-Modus müssen sowohl Leser als auch weitere Schreiber warten.
- Ist ein Schreiber fertig, wird das Lock wieder frei gegeben...

Implementierung

```
1  public class RW {
2      private int countReaders = 0;
3
4      public synchronized void startRead ()
5                           throws InterruptedException {
6          while (countReaders < 0) wait ();
7          countReaders++;
8      }
9
10     public synchronized void endRead () {
11         countReaders--;
12         if (countReaders == 0) notifyAll ();
13     }
14 //continued...
```

Implementierung

```
15      public synchronized void startWrite ()
16                              throws InterruptedException {
17          while (countReaders != 0) wait ();
18          countReaders = -1;
19      }
20
21      public synchronized void endWrite () {
22          countReaders = 0;
23          notifyAll ();
24      }
25  }
```


Implementierung

```
1  public class RW {
2      private int countReaders = 0;
3
4      public synchronized void startRead ()
5                              throws InterruptedException {
6          while (countReaders < 0) wait ();
7          countReaders++;
8      }
9
10     public synchronized void endRead () {
11         countReaders--;
12         if (countReaders == 0) notifyAll ();
13     }
14 //continued...
```

Implementierung

```
15      public synchronized void startWrite ()
16                              throws InterruptedException {
17          while (countReaders != 0) wait ();
18          countReaders = -1;
19      }
20
21      public synchronized void endWrite () {
22          countReaders = 0;
23          notifyAll ();
24      }
25  }
```


Diskussion

- Die Methoden `startRead()`, und `endRead()` sollen eine Leseoperation eröffnen bzw. beenden.
- Die Methoden `startWrite()`, und `endWrite()` sollen eine Schreiboperation eröffnen bzw. beenden.
- Die Methoden sind `synchronized`, damit sie selbst atomar ausgeführt werden.
- Die unterschiedlichen Modi eines RW-Locks sind mit Hilfe des Zählers `count` implementiert.
- Ein negativer Zählerstand entspricht dem Schreib-Modus, während ein positiver Zählerstand die Anzahl der aktiven Leser bezeichnet. . .

Diskussion

- Die Methoden `startRead()`, und `endRead()` sollen eine Leseoperation eröffnen bzw. beenden.
- Die Methoden `startWrite()`, und `endWrite()` sollen eine Schreiboperation eröffnen bzw. beenden.
- Die Methoden sind `synchronized`, damit sie selbst atomar ausgeführt werden.
- Die unterschiedlichen Modi eines RW-Locks sind mit Hilfe des Zählers `count` implementiert.
- Ein negativer Zählerstand entspricht dem Schreib-Modus, während ein positiver Zählerstand die Anzahl der aktiven Leser bezeichnet...

Diskussion

- `startRead()` führt erst dann kein `wait()` aus, wenn das RW-Lock entweder frei oder im Lese-Modus ist.
  Dann wird der Zähler inkrementiert.
- `endRead()` dekrementiert den Zähler wieder.
  Ist danach das RW-Lock frei, werden alle wartenden Threads benachrichtigt.
- `startWrite()` führt erst dann kein `wait()` aus, wenn das RW-Lock definitiv frei ist.
  Dann wird der Zähler auf -1 gesetzt.
- `endWrite()` setzt den Zähler wieder auf 0 zurück und benachrichtigt dann alle wartenden Threads.

Diskussion

- `startRead()` führt erst dann kein `wait()` aus, wenn das RW-Lock entweder frei oder im Lese-Modus ist.
  Dann wird der Zähler inkrementiert.
- `endRead()` dekrementiert den Zähler wieder.
  Ist danach das RW-Lock frei, werden alle wartenden Threads benachrichtigt.
- `startWrite()` führt erst dann kein `wait()` aus, wenn das RW-Lock definitiv frei ist.
  Dann wird der Zähler auf -1 gesetzt.
- `endWrite()` setzt den Zähler wieder auf 0 zurück und benachrichtigt dann alle wartenden Threads.

Die HashTable mit RW-Lock

```
1  public class HashTable<K,V> {
2      private RW rw;
3      private List2<K,V> [] a;
4      private int n;
5      public HashTable (int n) {
6          rw = new RW();
7          a = new List2 [n];
8          this.n = n;
9      }
10 // continued...
```

Die HashTable mit RW-Lock

```
11      public V lookup (K key) throws
12                               InterruptedException {
13          rw.startRead();
14          int i = Math.abs(key.hashCode() % n);
15          V result = (a[i] == null) ? null :
16                                      a[i].lookup(key);
17          synchronized (rw) {
18              rw.endRead();
19              return result;
20          }
21      }
22  //continued
```

- Da `lookup()` nicht weiß, wie mit einem interrupt umzugehen ist, wird die Exception weiter geworfen.
- Damit zwischen Freigabe des RW-Locks und der Rückgabe des Ergebnisses kein Schreibvorgang möglich ist, wird die Rückgabe des RW-Locks mit dem `return` gekoppelt.

Die HashTable mit RW-Lock

```
1  public class HashTable<K,V> {
2      private RW rw;
3      private List2<K,V> [] a;
4      private int n;
5      public HashTable (int n) {
6          rw = new RW();
7          a = new List2 [n];
8          this.n = n;
9      }
10 // continued...
```

Die HashTable mit RW-Lock

```
23      public void update (K key, V value) throws
24                                  InterruptedException {
25          rw.startWrite();
26          int i = Math.abs(key.hashCode() % n);
27          if (a[i] == null)
28               a[i] = new List2<K,V> (key,value,null);
29          else
30               a[i].update(key,value);
31          rw.endWrite();
32      }
33  }
```

- Die Methode `update(K key, V value)` der Klasse `List2<K,V>` sucht nach Eintrag für `key`. Wird dieser gefunden, wird dort das Wert-Attribut auf `value` gesetzt.

  Andernfalls wird ein neues Listenobjekt für das Paar `(key,value)` angefügt.

Die HashTable mit RW-Lock

```
11      public V lookup (K key) throws
12                                  InterruptedException {
13          rw.startRead();
14          int i = Math.abs(key.hashCode() % n);
15          V result = (a[i] == null) ? null :
16                                          a[i].lookup(key);
17          synchronized (rw) {
18              rw.endRead();
19              return result;
20          }
21      }
22  //continued
```

- Da `lookup()` nicht weiß, wie mit einem interrupt umzugehen ist, wird die Exception weiter geworfen.
- Damit zwischen Freigabe des RW-Locks und der Rückgabe des Ergebnisses kein Schreibvorgang möglich ist, wird die Rückgabe des RW-Locks mit dem `return` gekoppelt.

Diskussion

- Die neue Implementierung unterstützt nebenläufige Lesezugriffe auf die HashTable.
- Ein einziger Lesezugriff blockiert aber Schreibzugriffe — selbst, wenn sie sich letztendlich auf andere Teillisten beziehen und damit unabhängig sind...
- Genauso blockiert ein einzelner Schreibzugriff sämtliche Lesezugriffe, selbst wenn sie sich auf andere Teillisten beziehen...

$\Longrightarrow$ Eingrenzung der kritischen Abschnitte...

Die HashTable mit RW-Lock

```
23      public void update (K key, V value) throws
24                                  InterruptedException {
25          rw.startWrite();
26          int i = Math.abs(key.hashCode() % n);
27          if (a[i] == null)
28               a[i] = new List2<K,V> (key,value,null);
29          else
30               a[i].update(key,value);
31          rw.endWrite();
32      }
33  }
```

- Die Methode `update(K key, V value)` der Klasse `List2<K,V>` sucht nach Eintrag für `key`. Wird dieser gefunden, wird dort das Wert-Attribut auf `value` gesetzt.

  Andernfalls wird ein neues Listenobjekt für das Paar `(key,value)` angefügt.

Realisierung

```
1  class ListHead<K,V> {
2      private List2<K,V> list = null;
3      private RW rw = new RW();
4      public V lookup (K key) throws InterruptedException {
5          rw.startRead();
6          V result= (list==null) ? null : list.lookup(key);
7          synchronized (rw) {
8              rw.endRead();
9              return result;
10     }   }
11     public void update (K key, V value) throws
12                          InterruptedException {
13         rw.startWrite();
14         if (list == null)
15             list = new List2<K,V>(key,value,null);
16         else list.update(key,value);
17         rw.endWrite();
18 }   }
```

Diskussion

- Die neue Implementierung unterstützt nebenläufige Lesezugriffe auf die HashTable.
- Ein einziger Lesezugriff blockiert aber Schreibzugriffe — selbst, wenn sie sich letztendlich auf andere Teillisten beziehen und damit unabhängig sind...
- Genauso blockiert ein einzelner Schreibzugriff sämtliche Lesezugriffe, selbst wenn sie sich auf andere Teillisten beziehen...

$\Longrightarrow$ Eingrenzung der kritischen Abschnitte...

Diskussion

- Jedes Objekt der Klasse `ListHead` enthält ein eigenes RW-Lock zusammen mit einer Liste (eventuell `null`).
- Die Methoden `lookup()` und `update` wählen erst (unsynchronisiert) die richtige Liste aus, um dann geordnet auf die ausgewählte Liste zuzugreifen...

```
// in der Klasse HashTable:
public V lookup (K key) throws InterruptedException {
    int i = Math.abs(key.hashCode() % n);
    return a[i].lookup(key);
}
public void update (K key, V value)
                    throws InterruptedException {
    int i = Math.abs(key.hashCode() % n);
    a[i].update (key, value);
}
```

Realisierung

```
1  class ListHead<K,V> {
2      private List2<K,V> list = null;
3      private RW rw = new RW();
4      public V lookup (K key) throws InterruptedException {
5          rw.startRead();
6          V result= (list==null) ? null : list.lookup(key);
7          synchronized (rw) {
8              rw.endRead();
9              return result;
10     }   }
11     public void update (K key, V value) throws
12                             InterruptedException {
13         rw.startWrite();
14         if (list == null)
15             list = new List2<K,V>(key,value,null);
16         else list.update(key,value);
17         rw.endWrite();
18 }   }
```

Lock Support in Java

- `ReentrantLock`
  `lock()`, `unlock()`, `tryLock()`, `newCondition()`

  `wait()`, `notify()` heißen `await()` und `signal()` und werden auf einem `Condition`-Objekt ausgeführt.
- Ein ReentrantLock kann in einer anderen Funktion/einem anderen Block (anders als `synchronized`) freigegeben werden.
- **Achtung**: Ein Reentrant-Lock kann nicht von einem anderen Thread freigegeben werden.

Diskussion

- Jedes Objekt der Klasse `ListHead` enthält ein eigenes RW-Lock zusammen mit einer Liste (eventuell `null`).
- Die Methoden `lookup()` und `update` wählen erst (unsynchronisiert) die richtige Liste aus, um dann geordnet auf die ausgewählte Liste zuzugreifen...

```
// in der Klasse HashTable:
public V lookup (K key) throws InterruptedException {
    int i = Math.abs(key.hashCode() % n);
    return a[i].lookup(key);
}
public void update (K key, V value)
                    throws InterruptedException {
    int i = Math.abs(key.hashCode() % n);
    a[i].update (key, value);
}
```

Lock Support in Java

```
1 class Test {
2     synchronized void put() throws
3                               InterruptedException {
4         ...
5         while (...) wait();
6         ...
7     }
8 }
```


Lock Support in Java

- `ReentrantLock`
  `lock()`, `unlock()`, `tryLock()`, `newCondition()`

  `wait()`, `notify()` heißen `await()` und `signal()` und werden auf einem `Condition`-Objekt ausgeführt.
- Ein ReentrantLock kann in einer anderen Funktion/einem anderen Block (anders als `synchronized`) freigegeben werden.
- **Achtung**: Ein Reentrant-Lock kann nicht von einem anderen Thread freigegeben werden.

Lock Support in Java

```
class Test {
    synchronized void put() throws
                                InterruptedException {
        ...
        while (...) wait();
        ...
    }
}
```


Lock Support in Java

```
class Test {
    Lock lock = new ReentrantLock();
    Condition cond = lock.newCondition();
    void put() {
        lock.lock();
        try {
            ...
            while (...) cond.await()
            ...
        }
        finally {
            lock.unlock();
        }
    }
}
```

Lock Support in Java

```
1  class Test {
2      Lock lock = new ReentrantLock();
3      Condition cond = lock.newCondition();
4      void put() {
5          lock.lock();
6          try {
7              ...
8              while (...) cond.await()
9              ...
10         }
11         finally {
12             lock.unlock();
13         }
14     }
15 }
```


Semaphore Support in Java

- Klasse `Semaphore`,
- Konstruktor `Semaphore(int permits)`
- Hauptmethoden: `acquire()`, `release()`

Kann von einem anderen Thread `release()`d werden als von dem, der `acquire()` aufgerufen hat.

Semaphore Support in Java

- Klasse `Semaphore`,
- Konstruktor `Semaphore(int permits)`
- Hauptmethoden: `acquire()`, `release()`

Kann von einem anderen Thread `release()`d werden als von dem, der `acquire()` aufgerufen hat.

RW-Locks in Java

- Klasse `ReentrantReadWriteLock`
- `readLock()`, `writeLock` gibt das zugehörige readLock, bzw. writeLock zurück.

```
1 ...
2 ReentrantReadWriteLock rw;
3 rw.readLock().lock();
4 try {
5   ...
6 }
7 finally { rw.readLock().unlock(); }
8 ...
```

RW-Locks in Java

- Klasse `ReentrantReadWriteLock`
- `readLock()`, `writeLock` gibt das zugehörige readLock, bzw. writeLock zurück.

```
1 ...
2 ReentrantReadWriteLock rw;
3 rw.readLock().lock();
4 try {
5   ...
6 }
7 finally { rw.readLock().unlock(); }
8 ...
```


Warnung

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

- ... wenige Threads zu erzeugen als mehr.
- ... unabhängige Threads zu erzeugen als sich wechselseitig beeinflussende.
- ... kritische Abschnitte zu schützen, als nicht synchronisierte Operationen zu erlauben.
- ... kritsche Abschnitte zu entkoppeln, als sie zu schachteln.

Warnung

Finden der Fehler bzw. Überprüfung der Korrektheit ist ungleich schwieriger als für sequentielle Programme:

- Fehlerhaftes Verhalten tritt eventuell nur gelegentlich auf...
- bzw. nur für bestimmte Scheduler.
- Die Anzahl möglicher Programm-Ausführungsfolgen mit potentiell unterschiedlichem Verhalten ist gigantisch.
- Heisenbugs.

Warnung

Threads sind nützlich, sollten aber nur mit Vorsicht eingesetzt werden. Es ist besser,

- ...wenige Threads zu erzeugen als mehr.
- ...unabhängige Threads zu erzeugen als sich wechselseitig beeinflussende.
- ...kritische Abschnitte zu schützen, als nicht synchronisierte Operationen zu erlauben.
- ...kritsche Abschnitte zu entkoppeln, als sie zu schachteln.

17 Graphische Benutzeroberflächen

Eine graphische Benutzer-Oberfläche (GUI) ist i.A. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich) intuitiven Dialog mit der Benutzerin ermöglichen sollen.

Idee:

- Einzelne Komponenten bieten der Benutzerin Aktionen an.
- Ausführen der Aktionen erzeugt Ereignisse.
- Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht Ereignis-basiertes Programmieren.

17 Graphische Benutzeroberflächen

Eine graphische Benutzer-Oberfläche (GUI) ist i.A. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich) intuitiven Dialog mit der Benutzerin ermöglichen sollen.

Idee:

- Einzelne Komponenten bieten der Benutzerin Aktionen an.
- Ausführen der Aktionen erzeugt Ereignisse.
- Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht Ereignis-basiertes Programmieren.

Ereignisse

Ereignisse

- Maus-Bewegungen und -Klicks, Tastatureingaben etc. werden von der Peripherie registriert und an das ↑Betriebssystem weitergeleitet.
- Das Java-Laufzeitsystem nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür `AWTEvent`-Objekte.
- Diese Objekte werden in eine `AWTEventQueue` eingetragen Producer!
- Die Ereignisschlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen. . .
- Der `AWTEvent`-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet Consumer!

Ereignisse

Ereignisse

- Maus-Bewegungen und -Klicks, Tastatureingaben etc. werden von der Peripherie registriert und an das ↑Betriebssystem weitergeleitet.
- Das Java-Laufzeitsystem nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür AWTEvent-Objekte.
- Diese Objekte werden in eine AWTEventQueue eingetragen Producer!
- Die Ereignisschlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen...
- Der AWTEvent-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet Consumer!

Ereignisse

- Abarbeiten eines Ereignisses bedeutet:
  1. Weiterleiten des AWTEvent-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse angemeldet wurde;
  2. Aufrufen einer speziellen Methode des Listener-Objekts.
- Die Objekt-Methode des Listener-Objekts hat für die Reaktion des Applets zu sorgen.

GUI-Frameworks

AWT, Abstract Windowing Toolkit.

- nutzt GUI-Elemente des Betriebssystems
- gut für Effizienz
- Anwendungen sehen auf verschiedenen Systemen unterschiedlich aus (kann Vorteil aber auch Nachteil sein)
- unterstützt üblicherweise nur Elemente die auf den meisten Systemen verfügbar sind
- funktioniert mit Applets

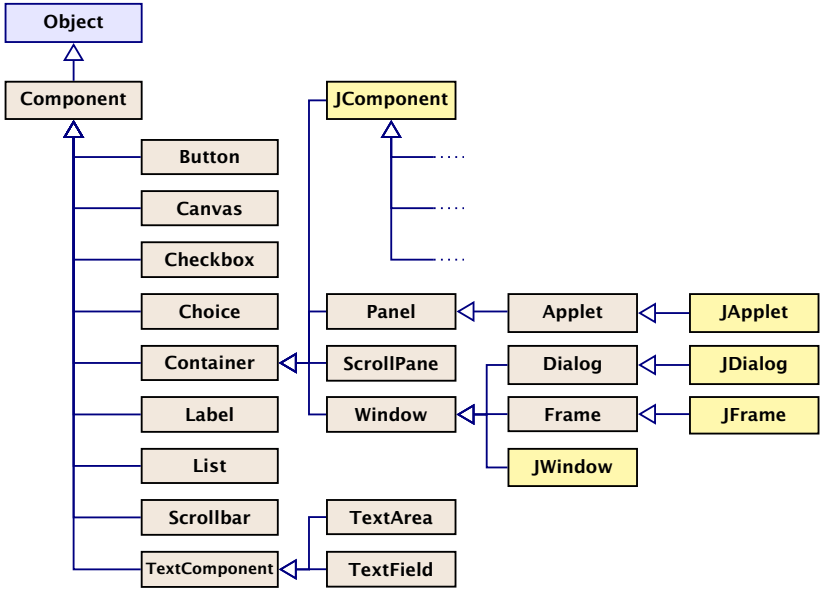
Swing

- fast alle GUI-Elemente sind in Java implementiert
- Anwendungen sehen überall gleich aus; (aber skinnable)
- reichhaltigere Sammlung von Elementen

Ereignisse

- Abarbeiten eines Ereignisses bedeutet:
  1. Weiterleiten des `AWTEvent`-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse angemeldet wurde;
  2. Aufrufen einer speziellen Methode des Listener-Objekts.
- Die Objekt-Methode des Listener-Objekts hat für die Reaktion des Applets zu sorgen.

Elemente in AWT und Swing



GUI-Frameworks

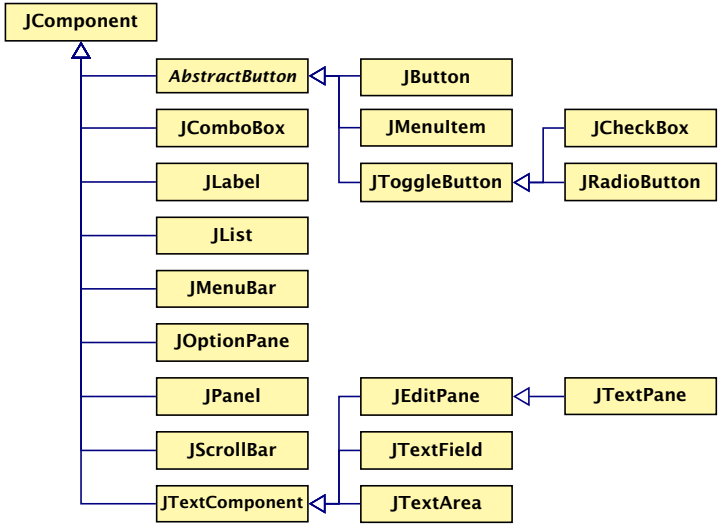
AWT, Abstract Windowing Toolkit.

- nutzt GUI-Elemente des Betriebssystems
- gut für Effizienz
- Anwendungen sehen auf verschiedenen Systemen unterschiedlich aus (kann Vorteil aber auch Nachteil sein)
- unterstützt üblicherweise nur Elemente die auf den meisten Systemen verfügbar sind
- funktioniert mit Applets

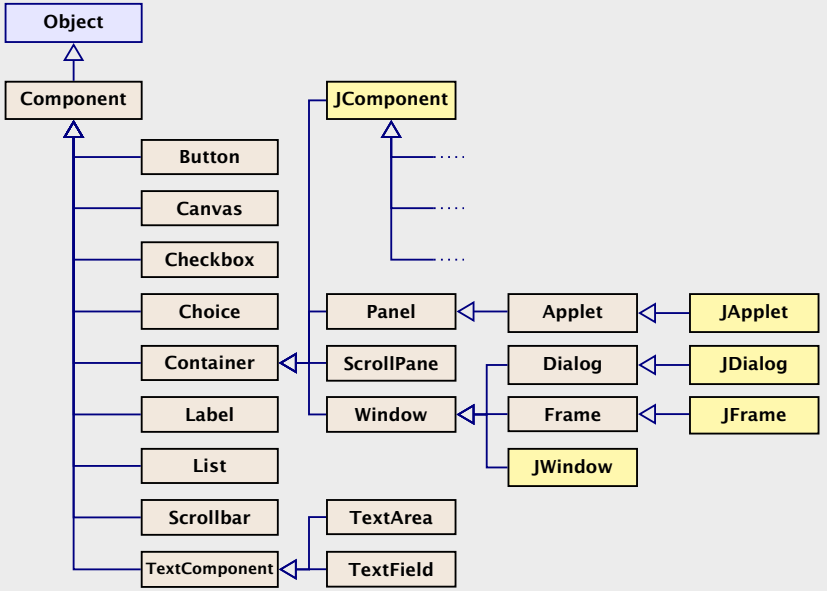
Swing

- fast alle GUI-Elemente sind in Java implementiert
- Anwendungen sehen überall gleich aus; (aber skinnable)
- reichhaltigere Sammlung von Elementen

Elemente in AWT und Swing



Elemente in AWT und Swing

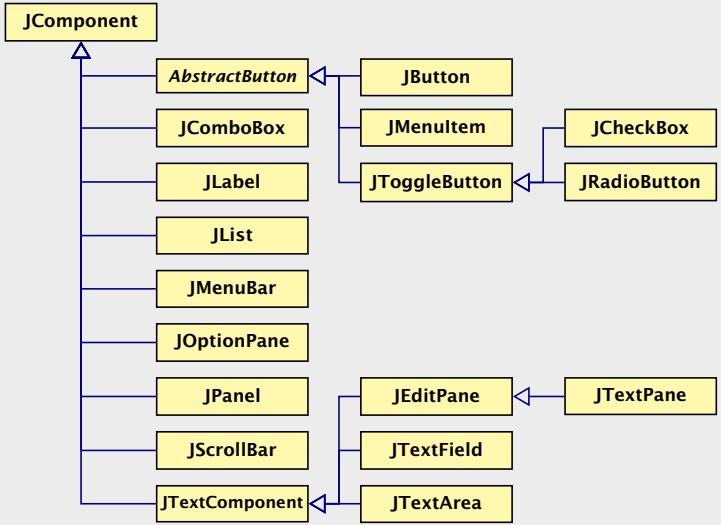


Ein Button

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  public class FirstButton extends JFrame implements
4                                          ActionListener {
5      JLabel label;
6      JButton button;
7      public FirstButton() {
8          setLayout(new FlowLayout());
9          setSize(500,100);
10         setVisible(true);
11         setFont(new Font("SansSerif", Font.BOLD, 18));
12         label = new JLabel();
13         label.setText("This is my first button :-)");
14         add(label);
15         button = new JButton("Knopf");
16         button.addActionListener(this);
17         add(button);
18         revalidate();
19     }
```

"FirstButton.java"

Elemente in AWT und Swing



Ein Button

```
20      public void actionPerformed(ActionEvent e) {
21          label.setText("Damn - you pressed it ...");
22          System.out.println(e);
23          remove(button);
24          // layout manager recalculates positions
25          revalidate();
26          repaint();
27      }
28      static class MyRunnable implements Runnable {
29          public void run() {
30              new FirstButton();
31          }
32      }
33      public static void main(String args[]) {
34          SwingUtilities.invokeLater(new MyRunnable());
35      }
36  } // end of FirstButton
```

"FirstButton.java"

Ein Button

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  public class FirstButton extends JFrame implements
4                                          ActionListener {
5      JLabel label;
6      JButton button;
7      public FirstButton() {
8          setLayout(new FlowLayout());
9          setSize(500,100);
10         setVisible(true);
11         setFont(new Font("SansSerif", Font.BOLD, 18));
12         label = new JLabel();
13         label.setText("This is my first button :-)");
14         add(label);
15         button = new JButton("Knopf");
16         button.addActionListener(this);
17         add(button);
18         revalidate();
19     }
```

"FirstButton.java"

Erläuterungen

- Wir erzeugen einen JFrame; ein normales Fenster mit Menüleiste, etc.
- Wir setzen Größe (setSize) des Frames, und machen ihn sichtbar (setVisible).
- setLayout kommt später...
- Der Frame enthält zwei weitere Komponenten:
  - ein JButton
  - ein JLabel
- Objekte dieser Klassen besitzen eine Aufschrift...
- Die in den Labels verwendete Schriftart richtet sich nach der des umgebenden Containers (zumindest in der Größe); deshalb wählen wir eine Schrift für den Frame

Ein Button

```
20      public void actionPerformed(ActionEvent e) {
21          label.setText("Damn - you pressed it ...");
22          System.out.println(e);
23          remove(button);
24          // layout manager recalculates positions
25          revalidate();
26          repaint();
27      }
28      static class MyRunnable implements Runnable {
29          public void run() {
30              new FirstButton();
31          }
32      }
33      public static void main(String args[]) {
34          SwingUtilities.invokeLater(new MyRunnable());
35      }
36  } // end of FirstButton
```

"FirstButton.java"

Erläuterungen

- Die Objekt-Methoden:
  ```
  void add(Component c)
  void add(Component c, int i)
  ```
  ... fügen die Komponente `c` zum Container `JFrame` hinten (bzw. an der Stelle `i`) hinzu.
- `public void addActionListener(ActionListener listener)` registriert ein Objekt `listener` als das, welches die von der Komponente ausgelösten `ActionEvent`-Objekte behandelt, hier: der `JFrame` selber.
- `ActionListener` ist ein Interface. Für die Implementierung muss die Methode `void actionPerformed(ActionEvent e)` bereitgestellt werden.

Erläuterungen

- Wir erzeugen einen `JFrame`; ein normales Fenster mit Menüleiste, etc.
- Wir setzen Größe (`setSize`) des Frames, und machen ihn sichtbar (`setVisible`).
- `setLayout` kommt später...
- Der Frame enthält zwei weitere Komponenten:
  - ein `JButton`
  - ein `JLabel`
- Objekte dieser Klassen besitzen eine Aufschrift...
- Die in den Labels verwendete Schriftart richtet sich nach der des umgebenden Containers (zumindest in der Größe); deshalb wählen wir eine Schrift für den Frame

Erläuterungen

- Die Methode `actionPerformed(ActionEvent e)` ersetzt den Text des Labels und entfernt den Knopf mithilfe der Methode `remove(Component c)`; anschließend muss der Container validiert und ggf. neu gezeichnet werden.
- Beim Drücken des Knopfs passiert das Folgende:
  1. ein `ActionEvent`-Objekt `action` wird erzeugt und in die Ereignisschlange eingefügt.
  2. Der `AWTEvent`-Dispatcher holt `action` wieder aus der Schlange. Er identifiziert den Frame `f` selbst als das für `action` zuständige Listener-Objekt. Darum ruft er `f.actionPerformed(action);` auf.
- Wären mehrere Objekte als `listener` registriert worden, würden sukzessive auch für diese entsprechende Aufrufe abgearbeitet werden.

Erläuterungen

- Die Objekt-Methoden:

  ```
  void add(Component c)
  void add(Component c, int i)
  ```

  ... fügen die Komponente `c` zum Container `JFrame` hinten (bzw. an der Stelle `i`) hinzu.
- `public void addActionListener(ActionListener listener)` registriert ein Objekt `listener` als das, welches die von der Komponente ausgelösten `ActionEvent`-Objekte behandelt, hier: der `JFrame` selber.
- `ActionListener` ist ein Interface. Für die Implementierung muss die Methode `void actionPerformed(ActionEvent e)` bereitgestellt werden.

Ein Button

Erläuterungen

- Die Methode `actionPerformed(ActionEvent e)` ersetzt den Text des Labels und entfernt den Knopf mithilfe der Methode `remove(Component c)`; anschließend muss der Container validiert und ggf. neu gezeichnet werden.
- Beim Drücken des Knopfs passiert das Folgende:
  1. ein `ActionEvent`-Objekt `action` wird erzeugt und in die Ereignisschlange eingefügt.
  2. Der `AWTEvent`-Dispatcher holt `action` wieder aus der Schlange. Er identifiziert den Frame `f` selbst als das für `action` zuständige Listener-Objekt. Darum ruft er `f.actionPerformed(action);` auf.
- Wären mehrere Objekte als `listener` registriert worden, würden sukzessive auch für diese entsprechende Aufrufe abgearbeitet werden.

Mehrere Knöpfe

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
4
5 public class SeveralButtons extends JFrame implements
6                                         ActionListener {
7     JLabel label;
8     JButton butA, butB;
```

"SeveralButtons.java"

Ein Button

Mehrere Knöpfe

```
 9      public SeveralButtons() {
10          setLayout(new FlowLayout());
11          setSize(500,100);
12          setVisible(true);
13          setFont(new Font("SansSerif", Font.BOLD, 18));
14          label = new JLabel();
15          label.setText("Press key...");
16          add(label);
17          butA = new JButton("Knopf A");
18          butA.setActionCommand("1");
19          butA.addActionListener(this);
20          add(butA);
21          butB = new JButton("Knopf B");
22          butB.setActionCommand("2");
23          butB.addActionListener(this);
24          add(butB);
25      }
```

"SeveralButtons.java"

Mehrere Knöpfe

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class SeveralButtons extends JFrame implements
6                                              ActionListener {
7      JLabel label;
8      JButton butA, butB;
```

"SeveralButtons.java"

Mehrere Knöpfe

```
26      public void actionPerformed(ActionEvent e) {
27          if (e.getActionCommand().equals("1")) {
28              label.setText("Toll! Knopf 1 mit Label "+
29                    ((JButton)e.getSource()).getText());
30          } else {
31              label.setText("Toll! Knopf 2 mit Label "+
32                    ((JButton)e.getSource()).getText());
33          }
34          System.out.println(e);
35      }
36      static class MyRunnable implements Runnable {
37          public void run() {
38              new SeveralButtons();
39          }
40      }
41      public static void main(String args[]) {
42          SwingUtilities.invokeLater(new MyRunnable());
43      }
44  }
```

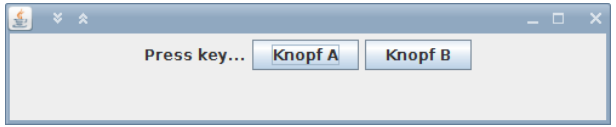
"SeveralButtons.java"

Mehrere Knöpfe

```
 9      public SeveralButtons() {
10          setLayout(new FlowLayout());
11          setSize(500,100);
12          setVisible(true);
13          setFont(new Font("SansSerif", Font.BOLD, 18));
14          label = new JLabel();
15          label.setText("Press key...");
16          add(label);
17          butA = new JButton("Knopf A");
18          butA.setActionCommand("1");
19          butA.addActionListener(this);
20          add(butA);
21          butB = new JButton("Knopf B");
22          butB.setActionCommand("2");
23          butB.addActionListener(this);
24          add(butB);
25      }
```

"SeveralButtons.java"

Mehrere Knöpfe



Mehrere Knöpfe

```
26      public void actionPerformed(ActionEvent e) {
27          if (e.getActionCommand().equals("1")) {
28              label.setText("Toll! Knopf 1 mit Label "+
29                    ((JButton)e.getSource()).getText());
30          } else {
31              label.setText("Toll! Knopf 2 mit Label "+
32                    ((JButton)e.getSource()).getText());
33          }
34          System.out.println(e);
35      }
36      static class MyRunnable implements Runnable {
37          public void run() {
38              new SeveralButtons();
39          }
40      }
41      public static void main(String args[]) {
42          SwingUtilities.invokeLater(new MyRunnable());
43      }
44  }
```

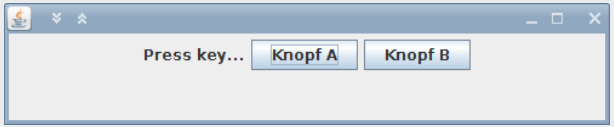
"SeveralButtons.java"

Alternativen

Wo kann man `EventListener` platzieren?

1. In der Klasse, die das Widget enthält (wie bei uns).
   - Widgets teilen sich Eventfunktionen (z.B. `ActionPerformed()`). Fallunterscheidung notwendig.
   - Die Widgets sind nicht von der Ereignisverarbeitung getrennt.
2. In einer/mehreren anderen Klasse.
   - Trennung von Ereignisverarbeitung und graphischen Elementen.
   - Bei einer Klasse Fallunterscheidungen erforderlich; mehrere Klassen führen evt. zu sehr viel Code
   - Zugriffe auf private Elemente?
3. Inner Class
4. Anonymous Inner Class

Mehrere Knöpfe



Inner Class

```
1 public class OuterClass {
2     private int var;
3     public class InnerClass {
4         void methodA() {};
5     }
6     public void methodB() {};
7 }
```

- Instanz von `InnerClass` kann auf alle Member von OuterClass zugreifen.
- Wenn `InnerClass` `static` deklariert wird, kann man nur auf statische Member zugreifen.
- Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.
- Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

Alternativen

Wo kann man `EventListener` platzieren?

1. In der Klasse, die das Widget enthält (wie bei uns).
   - Widgets teilen sich Eventfunktionen (z.B. `ActionPerformed()`). Fallunterscheidung notwendig.
   - Die Widgets sind nicht von der Ereignisverarbeitung getrennt.
2. In einer/mehreren anderen Klasse.
   - Trennung von Ereignisverarbeitung und graphischen Elementen.
   - Bei einer Klasse Fallunterscheidungen erforderlich; mehrere Klassen führen evt. zu sehr viel Code
   - Zugriffe auf private Elemente?
3. Inner Class
4. Anonymous Inner Class

Beispiel – Zugriff von Außen

```
1  class OuterClass {
2      private int x = 1;
3      public class InnerClass {
4          void show() {
5              System.out.println("x = " + x);
6          }
7      }
8      public void showMeth() {
9          InnerClass b = new InnerClass();
10         b.show();
11 }    }
12 public class TestInner {
13     public static void main(String args[]) {
14         OuterClass a = new OuterClass();
15         OuterClass.InnerClass x = a.new InnerClass();
16         x.show();
17         a.showMeth();
18 }    }
```

"TestInner.java"

Inner Class

```
1 public class OuterClass {
2     private int var;
3     public class InnerClass {
4         void methodA() {};
5     }
6     public void methodB() {};
7 }
```

- Instanz von `InnerClass` kann auf alle Member von OuterClass zugreifen.
- Wenn `InnerClass` `static` deklariert wird, kann man nur auf statische Member zugreifen.
- Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.
- Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

Beispiel – Zugriff von Außen

```
1  class OuterClass {
2      private static int x = 1;
3      public static class InnerClass {
4          void show() {
5              System.out.println("x = " + x);
6      }   }
7      public void showMeth() {
8          InnerClass b = new InnerClass();
9          b.show();
10 }   }
11 public class TestInnerStatic {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x =
15                   new OuterClass.InnerClass();
16         x.show();
17         a.showMeth();
18 }   }
```

"TestInnerStatic.java"

Beispiel – Zugriff von Außen

```
1  class OuterClass {
2      private int x = 1;
3      public class InnerClass {
4          void show() {
5              System.out.println("x = " + x);
6          }
7      }
8      public void showMeth() {
9          InnerClass b = new InnerClass();
10         b.show();
11 }   }
12 public class TestInner {
13     public static void main(String args[]) {
14         OuterClass a = new OuterClass();
15         OuterClass.InnerClass x = a.new InnerClass();
16         x.show();
17         a.showMeth();
18 }   }
```

"TestInner.java"

Local Inner Class

Eine lokale, innere Klasse wird innerhalb einer Methode deklariert:

```
1 public class OuterClass {
2     private int var;
3     public void methodA() {
4         class InnerClass {
5             void methodB() {};
6         }
7     }
8 }
```

- Kann zusätzlich auf die finalen Parameter und Variablen der Methode zugreifen.

Beispiel – Zugriff von Außen

```
1  class OuterClass {
2      private static int x = 1;
3      public static class InnerClass {
4          void show() {
5              System.out.println("x = " + x);
6      }   }
7      public void showMeth() {
8          InnerClass b = new InnerClass();
9          b.show();
10 }   }
11 public class TestInnerStatic {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x =
15                   new OuterClass.InnerClass();
16         x.show();
17         a.showMeth();
18 }   }
```

"TestInnerStatic.java"

Beispiel – Iterator

```
1 interface Iterator<T> {
2     boolean hasNext();
3     T next();
4     void remove(); // optional
5 }
```

- Ein Iterator erlaubt es über die Elemente einer Kollektion zu iterieren.
- Abstrahiert von der Implementierung der Kollektion.
- `hasNext()` testet, ob noch ein Element verfügbar ist.
- `next()` liefert das nächste Element (falls keins verfügbar ist wird eine `NoSuchElementException` geworfen).
- `remove()` entfernt das zuletzt über `next()` zugegriffene Element aus der Kollektion.

Local Inner Class

Eine lokale, innere Klasse wird innerhalb einer Methode deklariert:

```
1 public class OuterClass {
2     private int var;
3     public void methodA() {
4         class InnerClass {
5             void methodB() {};
6         }
7     }
8 }
```

- Kann zusätzlich auf die finalen Parameter und Variablen der Methode zugreifen.

Beispiel – Iterator

```
1  public class TestIterator {
2      Integer[] arr;
3      TestIterator(int n) {
4          arr = new Integer[n];
5      }
6      public Iterator<Integer> iterator() {
7          class MyIterator implements Iterator<Integer> {
8              int curr = arr.length;
9              public boolean hasNext() { return curr>0;}
10             public Integer next() {
11                 if (curr == 0)
12                     throw new NoSuchElementException();
13                 return arr[--curr];
14             }
15         }
16         return new MyIterator();
17     }
```

"TestIterator.java"

Beispiel – Iterator

```
1  interface Iterator<T> {
2      boolean hasNext();
3      T next();
4      void remove(); // optional
5  }
```

- Ein Iterator erlaubt es über die Elemente einer Kollektion zu iterieren.
- Abstrahiert von der Implementierung der Kollektion.
- `hasNext()` testet, ob noch ein Element verfügbar ist.
- `next()` liefert das nächste Element (falls keins verfügbar ist wird eine `NoSuchElementException` geworfen).
- `remove()` entfernt das zuletzt über `next()` zugegriffene Element aus der Kollektion.

Beispiel – Iterator

Anwendung des Iterators:

```
18      public static void main(String args[]) {
19          TestIterator t = new TestIterator(10);
20          Integer i = null;
21          for (Iterator<Integer> iter = t.iterator();
22               iter.hasNext(); i = iter.next()) {
23              System.out.println(i);
24          }
25      }
26  }
```

"TestIterator.java"

In diesem Fall wird nur 10 mal null ausgegeben...

Beispiel – Iterator

```
1  public class TestIterator {
2      Integer[] arr;
3      TestIterator(int n) {
4          arr = new Integer[n];
5      }
6      public Iterator<Integer> iterator() {
7          class MyIterator implements Iterator<Integer> {
8              int curr = arr.length;
9              public boolean hasNext() { return curr>0;}
10             public Integer next() {
11                 if (curr == 0)
12                     throw new NoSuchElementException();
13                 return arr[--curr];
14             }
15         }
16         return new MyIterator();
17     }
```

"TestIterator.java"

Anonymous Inner Classes

Der Anwendungsfall für lokale, innere Klassen ist häufig:

- eine Methode erzeugt genau ein Objekt der inneren Klasse
- dieses wird z.B. an den Aufrufer zurückgegeben

Anonyme Innere Klasse:

- Ausdruck enthält Klassendeklaration, und instanziiert ein Objekt der Klasse
- man gibt ein Interface an, dass implementiert wird, oder eine Klasse von der geerbt wird
- die Klasse selber erhält keinen Namen

Beispiel – Iterator

Anwendung des Iterators:

```
18      public static void main(String args[]) {
19          TestIterator t = new TestIterator(10);
20          Integer i = null;
21          for (Iterator<Integer> iter = t.iterator();
22               iter.hasNext(); i = iter.next()) {
23              System.out.println(i);
24          }
25      }
26  }
```

"TestIterator.java"

In diesem Fall wird nur 10 mal `null` ausgegeben...

Beispiel – Iterator

```
public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
        int curr = arr.length;
        public boolean hasNext() { return curr>0;}
        public Integer next() {
            if (curr == 0)
                throw new NoSuchElementException();
            return arr[--curr];
        }
    };
}
```

"IteratorAnonymous.java"

Anonymous Inner Classes

Der Anwendungsfall für lokale, innere Klassen ist häufig:

- eine Methode erzeugt genau ein Objekt der inneren Klasse
- dieses wird z.B. an den Aufrufer zurückgegeben

Anonyme Innere Klasse:

- Ausdruck enthält Klassendeklaration, und instanziiert ein Objekt der Klasse
- man gibt ein Interface an, dass implementiert wird, oder eine Klasse von der geerbt wird
- die Klasse selber erhält keinen Namen

Mehrere Knöpfe – Andere Klassse(n)

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3
4  class ListenerA implements ActionListener {
5      JLabel label;
6      ListenerA(JLabel l) { label = l; }
7      public void actionPerformed(ActionEvent e) {
8          label.setText("Toll! Knopf 1 mit Label "+
9                  ((JButton)e.getSource()).getText());
10 }    }
11 class ListenerB implements ActionListener {
12     JLabel label;
13     ListenerB(JLabel l) { label = l; }
14     public void actionPerformed(ActionEvent e) {
15         label.setText("Toll! Knopf 2 mit Label "+
16                 ((JButton)e.getSource()).getText());
17 }    }
```

"SeveralButtonsOther.java"

Beispiel – Iterator

```
public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
        int curr = arr.length;
        public boolean hasNext() { return curr>0;}
        public Integer next() {
            if (curr == 0)
                throw new NoSuchElementException();
            return arr[--curr];
        }
    };
}
```

"IteratorAnonymous.java"

Mehrere Knöpfe – Andere Klasse(n)

```
19 public class SeveralButtonsOther extends JFrame {
20     private JLabel label;
21     private JButton butA, butB;
22
23     public SeveralButtonsOther() {
24         setLayout(new FlowLayout());
25         setSize(500,100);
26         setVisible(true);
27         setFont(new Font("SansSerif", Font.BOLD, 18));
28         label = new JLabel();
29         label.setText("Press key...");
30         add(label);
31         butA = new JButton("Knopf A");
32         butA.addActionListener(new ListenerA(label));
33         add(butA);
34         butB = new JButton("Knopf B");
35         butB.addActionListener(new ListenerB(label));
36         add(butB);
37     }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Andere Klasse(n)

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3
4  class ListenerA implements ActionListener {
5      JLabel label;
6      ListenerA(JLabel l) { label = l; }
7      public void actionPerformed(ActionEvent e) {
8          label.setText("Toll! Knopf 1 mit Label "+
9                  ((JButton)e.getSource()).getText());
10 }    }
11 class ListenerB implements ActionListener {
12     JLabel label;
13     ListenerB(JLabel l) { label = l; }
14     public void actionPerformed(ActionEvent e) {
15         label.setText("Toll! Knopf 2 mit Label "+
16                 ((JButton)e.getSource()).getText());
17 }    }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Andere Klasse(n)

```
38      public static void main(String args[]) {
39          SwingUtilities.invokeLater(
40              new Runnable() {
41                  public void run() {
42                      new SeveralButtonsOther();
43                  }
44              }
45          );
46      }
47  }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Andere Klasse(n)

```
19  public class SeveralButtonsOther extends JFrame {
20      private JLabel label;
21      private JButton butA, butB;
22
23      public SeveralButtonsOther() {
24          setLayout(new FlowLayout());
25          setSize(500,100);
26          setVisible(true);
27          setFont(new Font("SansSerif", Font.BOLD, 18));
28          label = new JLabel();
29          label.setText("Press key...");
30          add(label);
31          butA = new JButton("Knopf A");
32          butA.addActionListener(new ListenerA(label));
33          add(butA);
34          butB = new JButton("Knopf B");
35          butB.addActionListener(new ListenerB(label));
36          add(butB);
37      }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Inner Class

```
1  import javax.swing.*;
2  import static javax.swing.SwingUtilities.*;
3  import java.awt.*; import java.awt.event.*;
4
5  public class SeveralButtonsInner extends JFrame {
6      private JLabel label;
7      private JButton butA, butB;
8      public class listenerA implements ActionListener {
9          public void actionPerformed(ActionEvent e) {
10             label.setText("Toll! Knopf 1 mit Label "+
11                    ((JButton)e.getSource()).getText());
12     }   }
13     public class listenerB implements ActionListener {
14         public void actionPerformed(ActionEvent e) {
15             label.setText("Toll! Knopf 2 mit Label "+
16                    ((JButton)e.getSource()).getText());
17     }   }
```

"SeveralButtonsInner.java"

Mehrere Knöpfe – Andere Klasse(n)

```
38     public static void main(String args[]) {
39         SwingUtilities.invokeLater(
40             new Runnable() {
41                 public void run() {
42                     new SeveralButtonsOther();
43                 }
44             }
45         );
46     }
47 }
```

"SeveralButtonsOther.java"

Mehrere Knöpfe – Inner Class

```
18      public SeveralButtonsInner() {
19          setLayout(new FlowLayout());
20          setSize(500,100);
21          setVisible(true);
22          setFont(new Font("SansSerif", Font.BOLD, 18));
23          label = new JLabel();
24          label.setText("Press key...");
25          add(label);
26          butA = new JButton("Knopf A");
27          butA.addActionListener(new listenerA());
28          add(butA);
29          butB = new JButton("Knopf B");
30          butB.addActionListener(new listenerB());
31          add(butB);
32      }
33      public static void main(String args[]) {
34          invokeLater(()->new SeveralButtonsInner());
35  }   }
```

"SeveralButtonsInner.java"

Mehrere Knöpfe – Inner Class

```
1  import javax.swing.*;
2  import static javax.swing.SwingUtilities.*;
3  import java.awt.*; import java.awt.event.*;
4
5  public class SeveralButtonsInner extends JFrame {
6      private JLabel label;
7      private JButton butA, butB;
8      public class listenerA implements ActionListener {
9          public void actionPerformed(ActionEvent e) {
10             label.setText("Toll! Knopf 1 mit Label "+
11                     ((JButton)e.getSource()).getText());
12     }   }
13     public class listenerB implements ActionListener {
14         public void actionPerformed(ActionEvent e) {
15             label.setText("Toll! Knopf 2 mit Label "+
16                     ((JButton)e.getSource()).getText());
17     }   }
```

"SeveralButtonsInner.java"

Mehrere Knöpfe – Anonymous Class

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3  import static javax.swing.SwingUtilities.*;
4  public class SeveralButtonsAnonymous extends JFrame {
5      JLabel label;
6      JButton butA, butB;
7      public static void main(String args[]) {
8          invokeLater(()->new SeveralButtonsAnonymous());
9      }
10     public SeveralButtonsAnonymous() {
11         setLayout(new FlowLayout());
12         setSize(500,100);
13         setVisible(true);
14         setFont(new Font("SansSerif", Font.BOLD, 18));
15         label = new JLabel();
16         label.setText("Press key...");
```

"SeveralButtonsAnonymous.java"

Mehrere Knöpfe – Inner Class

```
18     public SeveralButtonsInner() {
19         setLayout(new FlowLayout());
20         setSize(500,100);
21         setVisible(true);
22         setFont(new Font("SansSerif", Font.BOLD, 18));
23         label = new JLabel();
24         label.setText("Press key...");
25         add(label);
26         butA = new JButton("Knopf A");
27         butA.addActionListener(new listenerA());
28         add(butA);
29         butB = new JButton("Knopf B");
30         butB.addActionListener(new listenerB());
31         add(butB);
32     }
33     public static void main(String args[]) {
34         invokeLater(()->new SeveralButtonsInner());
35 }   }
```

"SeveralButtonsInner.java"

Mehrere Knöpfe – Anonymous Class

```
17          add(label);
18          butA = new JButton("Knopf A");
19          butA.addActionListener(new ActionListener() {
20              public void actionPerformed(ActionEvent e) {
21                  label.setText("Toll! Knopf 1 mit Label "+
22                          ((JButton)e.getSource()).getText());
23              }
24          });
25          add(butA);
26          butB = new JButton("Knopf B");
27          butB.addActionListener(new ActionListener() {
28              public void actionPerformed(ActionEvent e) {
29                  label.setText("Toll! Knopf 2 mit Label "+
30                          ((JButton)e.getSource()).getText());
31              }
32          });
33          add(butB);
34  }   }
```

"SeveralButtonsAnonymous.java"

Mehrere Knöpfe – Anonymous Class

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3  import static javax.swing.SwingUtilities.*;
4  public class SeveralButtonsAnonymous extends JFrame {
5      JLabel label;
6      JButton butA, butB;
7      public static void main(String args[]) {
8          invokeLater(()->new SeveralButtonsAnonymous());
9      }
10     public SeveralButtonsAnonymous() {
11         setLayout(new FlowLayout());
12         setSize(500,100);
13         setVisible(true);
14         setFont(new Font("SansSerif", Font.BOLD, 18));
15         label = new JLabel();
16         label.setText("Press key...");
```

"SeveralButtonsAnonymous.java"

Diskussion

Für größere Projekte ist Variante 2 vorzuziehen, da sie kleinere Klassen erlaubt, und eine saubere Trennung zwischen Ereignisbehandlung und graphischer Ausgabe ermöglicht.

Der Umweg über Innere Klassen vermeidet Fallunterscheidungen aber macht den Code recht unübersichtlich.

Weitere Alternative: Lambda-Ausdrücke/Methodenreferenzen

Mehrere Knöpfe – Anonymous Class

```
17          add(label);
18          butA = new JButton("Knopf A");
19          butA.addActionListener(new ActionListener() {
20              public void actionPerformed(ActionEvent e) {
21                  label.setText("Toll! Knopf 1 mit Label "+
22                        ((JButton)e.getSource()).getText());
23              }
24          });
25          add(butA);
26          butB = new JButton("Knopf B");
27          butB.addActionListener(new ActionListener() {
28              public void actionPerformed(ActionEvent e) {
29                  label.setText("Toll! Knopf 2 mit Label "+
30                        ((JButton)e.getSource()).getText());
31              }
32          });
33          add(butB);
34  }   }
```

"SeveralButtonsAnonymous.java"

Lambda-Ausdrücke

Ein funktionales Interface ist ein Interface, das **genau** eine Methode enthält.

```
interface Runnable {
   void run();
}
```

Ein Lambda-Ausdruck ist das Literal eines Objektes, das ein funktionales Interface implementiert. Z.B.:

Syntax:

- allgemein
  `(%Parameterliste) -> {...}`
- nur `return`-statement/eine Anweisung (bei `void`-Funktion)
  `(%Parameterliste) -> %Ausdruck`
- nur genau ein Parameter
  `a -> {...}`

Diskussion

Für größere Projekte ist Variante 2 vorzuziehen, da sie kleinere Klassen erlaubt, und eine saubere Trennung zwischen Ereignisbehandlung und graphischer Ausgabe ermöglicht.

Der Umweg über Innere Klassen vermeidet Fallunterscheidungen aber macht den Code recht unübersichtlich.

Weitere Alternative: Lambda-Ausdrücke/Methodenreferenzen

Beispiele

```
Runnable r = () -> {System.out.println("Hello!");};
```

ist (im Prinzip) äquivalent zu

```
class Foo implements Runnable {
    void run() {
        System.out.println("Hello!");
    }
}
Runnable r = new Foo();
```

Lambda-Ausdrücke

Ein funktionales Interface ist ein Interface, das **genau** eine Methode enthält.

```
interface Runnable {
   void run();
}
```

Ein Lambda-Ausdruck ist das Literal eines Objektes, das ein funktionales Interface implementiert. Z.B.:

Syntax:

- allgemein
  `(%Parameterliste) -> {...}`
- nur `return`-statement/eine Anweisung (bei `void`-Funktion)
  `(%Parameterliste) -> %Ausdruck`
- nur genau ein Parameter
  `a -> {...}`

Beispiele

```java
interface Func<T> {
    T func(int arg);
}
public class Eval<T> {
    void eval(Func<T> f, int[] arr, T[] res) {
        for (int i=0; i<arr.length; i++) {
            res[i] = f.func(arr[i]);
        }
    }
    public static void main(String args[]) {
        int[] a = {1,2,3,4,5};
        Integer[] b = new Integer[5];
        new Eval<Integer>().eval(x->x*x, a, b);
        for (int i=0; i<5; i++) {
            System.out.print(b[i]+",");
        }
    }
}
```

"Eval.java"

Beispiele

```java
Runnable r = () -> {System.out.println("Hello!");};
```

ist (im Prinzip) äquivalent zu

```java
class Foo implements Runnable {
    void run() {
        System.out.println("Hello!");
    }
}
Runnable r = new Foo();
```

Beispiel – Überladen

```
1  interface Func1 {
2      String func(String arg);
3  }
4  interface Func2 {
5      int func(int arg);
6  }
7  interface Func3 {
8      String func(int arg);
9  }
10 public class Test {
11     static void foo(Func1 f) { }
12     static void foo(Func2 f) { }
13     static void foo(Func3 f) { }
14     public static void main(String args[]) {
15         foo(x->x);
16     }
17 }
```

"TestLambda.java"

Beispiele

```
1  interface Func<T> {
2      T func(int arg);
3  }
4  public class Eval<T> {
5      void eval(Func<T> f, int[] arr, T[] res) {
6          for (int i=0; i<arr.length; i++) {
7              res[i] = f.func(arr[i]);
8          }
9      }
10     public static void main(String args[]) {
11         int[] a = {1,2,3,4,5};
12         Integer[] b = new Integer[5];
13         new Eval<Integer>().eval(x->x*x, a, b);
14         for (int i=0; i<5; i++) {
15             System.out.print(b[i]+",");
16         }
17     }
18 }
```

"Eval.java"

Beispiele

```
Interface Block<T> {
    void apply(T t);
}
```

```
Interface Function<T> {
    T map (T t);
}
```

```
Function<Block<String>> twice
            = b -> t -> { b.apply(t); b.apply(t); }
Block<String> print2
            = twice.map(s -> {System.out.println(s);});
print2.apply("hello");
```

```
final List<String> list = new ArrayList<>();
Block<String> adder2
            = twice.map(s -> {list.add(s);});
adder2.apply("world");
System.out.println(list);
```

Beispiel – Überladen

```
1  interface Func1 {
2      String func(String arg);
3  }
4  interface Func2 {
5      int func(int arg);
6  }
7  interface Func3 {
8      String func(int arg);
9  }
10 public class Test {
11     static void foo(Func1 f) { }
12     static void foo(Func2 f) { }
13     static void foo(Func3 f) { }
14     public static void main(String args[]) {
15         foo(x->x);
16     }
17 }
```

"TestLambda.java"

Methodenreferenzen

An der Stelle, an der ein Lambda-Ausdruck möglich ist, kann man auch eine Methodenreferenz einer passenden Methode angeben.

Beispiel:

- Klasse `ClassA` verfügt über statische Methode `boolean less(int a, int b)`.
- Das Funktionsinterface `Iface` verlangt die Implementierung einer Funktion, die zwei `int`s nach `boolean` abbildet.
- Außerdem existiert Funktion `sort(int[] a, Iface x)`.
- Dann sortiert der Aufruf:

```
int[] arr = {5,8,7,2,11};
sort(arr, ClassA::less);
```

gemäß der durch `less` vorgegebenen Ordnung.

Beispiele

```
Interface Block<T> {
    void apply(T t);
}
```

```
Interface Function<T> {
    T map (T t);
}
```

```
Function<Block<String>> twice
            = b -> t -> { b.apply(t); b.apply(t); }
Block<String> print2
            = twice.map(s -> {System.out.println(s);});
print2.apply("hello");
```

```
final List<String> list = new ArrayList<>();
Block<String> adder2
            = twice.map(s -> {list.add(s);});
adder2.apply("world");
System.out.println(list);
```

Mehrere Knöpfe – Lambda/Methodenreferenz

```
1  import javax.swing.*;
2  import static javax.swing.SwingUtilities.*;
3  import java.awt.*; import java.awt.event.*;
4  class EventHandler {
5      private JLabel label;
6      EventHandler(JLabel l) { label = l; }
7      public void actionA(ActionEvent e) {
8          label.setText("Toll! Knopf 1 mit Label
               "+e.getActionCommand());
9      }
10     public void actionB(ActionEvent e) {
11         label.setText("Toll! Knopf 2 mit Label
               "+e.getActionCommand());
12     }
13 }
14 public class SeveralButtonsLambda extends JFrame {
15     JLabel  label;
16     JButton butA, butB;
17     EventHandler handler;
```

Methodenreferenzen

An der Stelle, an der ein Lambda-Ausdruck möglich ist, kann man auch eine Methodenreferenz einer passenden Methode angeben.

Beispiel:

- Klasse `ClassA` verfügt über statische Methode `boolean less(int a, int b)`.
- Das Funktionsinterface `Iface` verlangt die Implementierung einer Funktion, die zwei `int`s nach `boolean` abbildet.
- Außerdem existiert Funktion `sort(int[] a, Iface x)`.
- Dann sortiert der Aufruf:

```
int[] arr = {5,8,7,2,11};
sort(arr, ClassA::less);
```

gemäß der durch `less` vorgegebenen Ordnung.

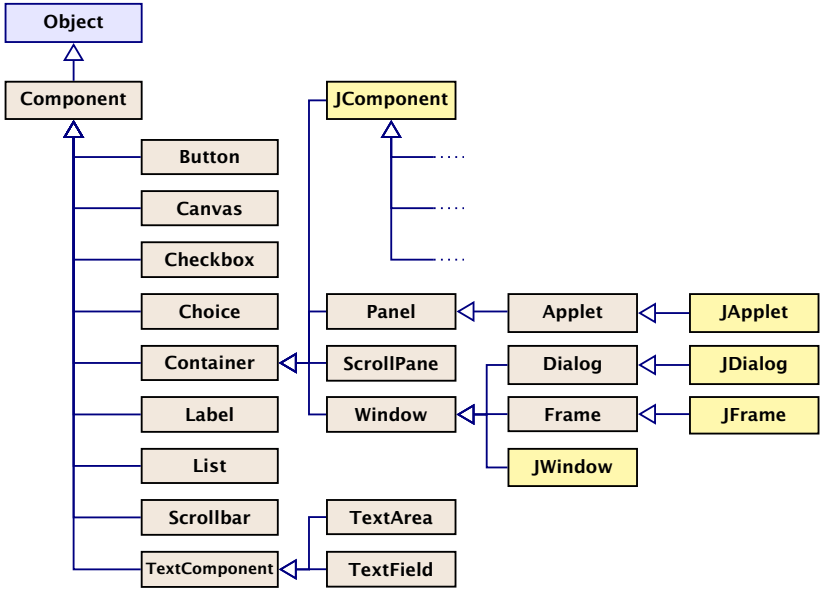
Mehrere Knöpfe – Lambda/Methodenreferenz

```
    public static void main(String args[]) {
        invokeLater(()->new SeveralButtonsLambda());
    }
    public SeveralButtonsLambda() {
        setLayout(new FlowLayout());
        setSize(500,100); setVisible(true);
        setFont(new Font("SansSerif", Font.BOLD, 18));
        label = new JLabel();
        label.setText("Press key..."); add(label);
        handler = new EventHandler(label);
        butA = new JButton("Knopf A");
        butA.addActionListener(handler::actionA);
        add(butA);
        butB = new JButton("Knopf B");
        butB.addActionListener(
           e -> label.setText("Toll! Knopf 2 mit Label "
            + e.getActionCommand()));
        add(butB);
}   }
```

Mehrere Knöpfe – Lambda/Methodenreferenz

```
import javax.swing.*;
import static javax.swing.SwingUtilities.*;
import java.awt.*; import java.awt.event.*;
class EventHandler {
    private JLabel label;
    EventHandler(JLabel l) { label = l; }
    public void actionA(ActionEvent e) {
        label.setText("Toll! Knopf 1 mit Label
            "+e.getActionCommand());
    }
    public void actionB(ActionEvent e) {
        label.setText("Toll! Knopf 2 mit Label
            "+e.getActionCommand());
    }
}
public class SeveralButtonsLambda extends JFrame {
    JLabel  label;
    JButton butA, butB;
    EventHandler handler;
```

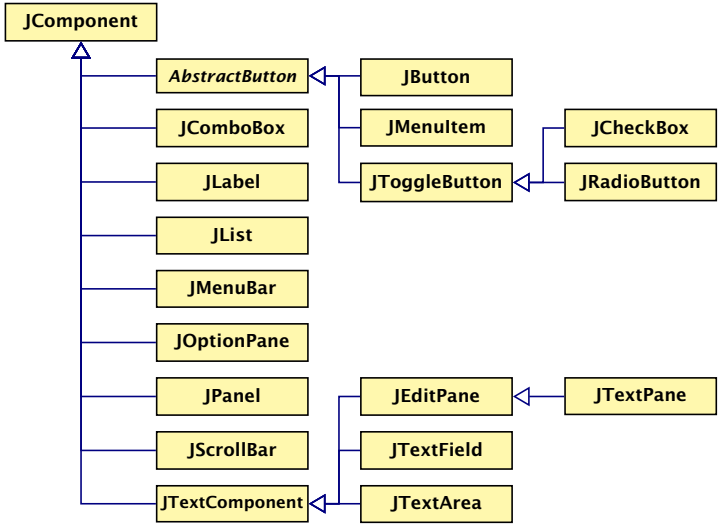
Elemente in AWT und Swing



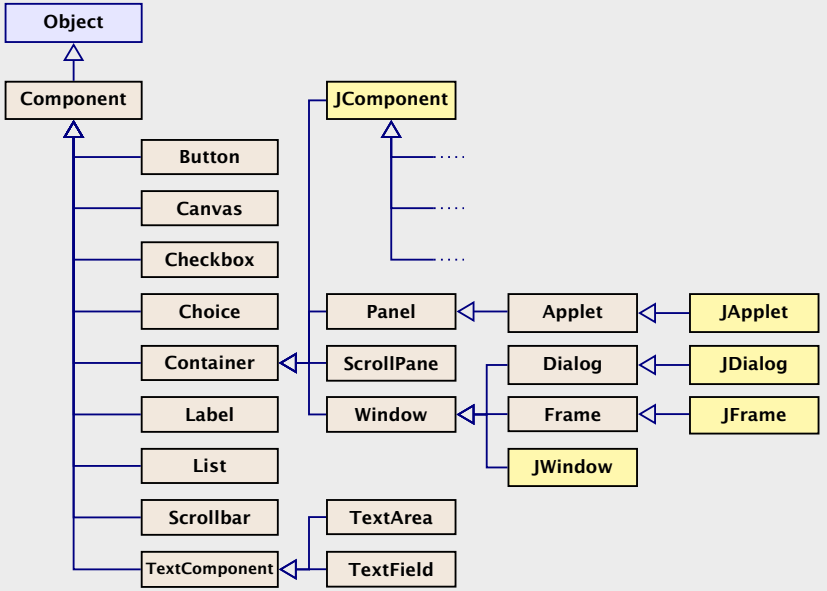
Mehrere Knöpfe – Lambda/Methodenreferenz

```
1      public static void main(String args[]) {
2          invokeLater(()->new SeveralButtonsLambda());
3      }
4      public SeveralButtonsLambda() {
5          setLayout(new FlowLayout());
6          setSize(500,100); setVisible(true);
7          setFont(new Font("SansSerif", Font.BOLD, 18));
8          label = new JLabel();
9          label.setText("Press key..."); add(label);
10         handler = new EventHandler(label);
11         butA = new JButton("Knopf A");
12         butA.addActionListener(handler::actionA);
13         add(butA);
14         butB = new JButton("Knopf B");
15         butB.addActionListener(
16             e -> label.setText("Toll! Knopf 2 mit Label "
17              + e.getActionCommand()));
18         add(butB);
19 }    }
```

Elemente in AWT und Swing



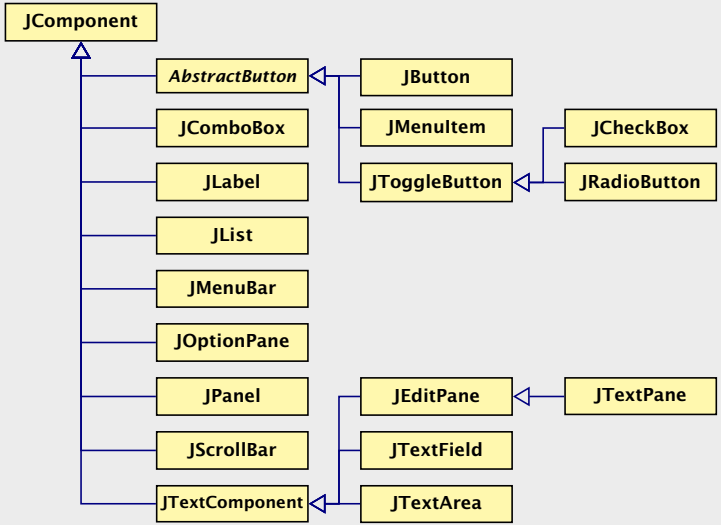
Elemente in AWT und Swing



Elemente:

(J)Label	Zeigt eine Textzeile.
(J)Button	Einzelner Knopf um Aktion auszulösen.
(J)Scrollbar	Schieber zur Eingabe kleiner `int`-Zahlen.

Elemente in AWT und Swing



Beispiel – Scrollbar

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3  public class ScalingFun extends JFrame
4                          implements AdjustmentListener {
5      private JScrollBar scrH, scrW;
6      private JComponent content;
7      private Image image;
8      private int width, height;
9
10     class MyComponent extends JComponent {
11         MyComponent(int w, int h) {
12             setPreferredSize(new Dimension(w,h));
13         }
14         public void paintComponent(Graphics page) {
15             int l = getWidth()/2  - width/2;
16             int r = getHeight()/2 - height/2;
17             page.drawImage(image,l,r,width,height,null);
18     }   }
```

"ScalingFun.java"

Elemente:

(J)Label	Zeigt eine Textzeile.
(J)Button	Einzelner Knopf um Aktion auszulösen.
(J)Scrollbar	Schieber zur Eingabe kleiner `int`-Zahlen.

Beispiel – Scrollbar

```
ScalingFun() { // Konstruktor
    image = Toolkit.getDefaultToolkit().
                        getImage("al-Chwarizmi.png");
    // wait for image to load...
    while (image.getHeight(null) == -1);
    int h = height = image.getHeight(null);
    int w = width  = image.getWidth(null);
    setLayout(new BorderLayout());
    scrH=new JScrollBar(JScrollBar.VERTICAL,h,50,0,h+50);
    scrH.addAdjustmentListener(this);
    add(scrH,"West");
    scrW=new JScrollBar(JScrollBar.HORIZONTAL,w,50,0,w+50);
    scrW.addAdjustmentListener(this);
    add(scrW,"South");
    setVisible(true);
    add(content = new MyComponent(w,h));
    pack();
}
```

"ScalingFun.java"

Beispiel – Scrollbar

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3  public class ScalingFun extends JFrame
4                          implements AdjustmentListener {
5      private JScrollBar scrH, scrW;
6      private JComponent content;
7      private Image image;
8      private int width, height;
9
10     class MyComponent extends JComponent {
11         MyComponent(int w, int h) {
12             setPreferredSize(new Dimension(w,h));
13         }
14         public void paintComponent(Graphics page) {
15             int l = getWidth()/2  - width/2;
16             int r = getHeight()/2 - height/2;
17             page.drawImage(image,l,r,width,height,null);
18     }   }
```

"ScalingFun.java"

Beispiel – Scrollbar

```
    public void adjustmentValueChanged(AdjustmentEvent e) {
        Adjustable s = e.getAdjustable();
        int value = e.getValue();
        if (s == scrH) height = value;
        else width = value;
        revalidate();
        repaint();
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new ScalingFun());
    }
} // end of ScalingFun
```

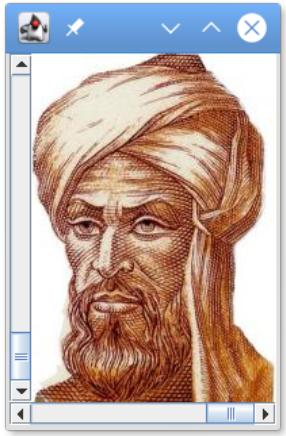
"ScalingFun.java"

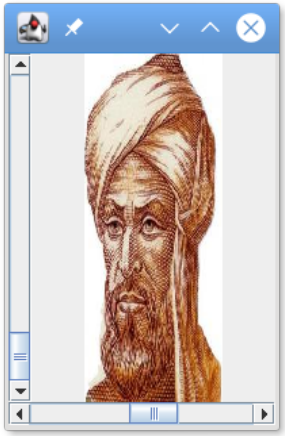
Beispiel – Scrollbar

```
ScalingFun() { // Konstruktor
    image = Toolkit.getDefaultToolkit().
                        getImage("al-Chwarizmi.png");
    // wait for image to load...
    while (image.getHeight(null) == -1);
    int h = height = image.getHeight(null);
    int w = width  = image.getWidth(null);
    setLayout(new BorderLayout());
    scrH=new JScrollBar(JScrollBar.VERTICAL,h,50,0,h+50);
    scrH.addAdjustmentListener(this);
    add(scrH,"West");
    scrW=new JScrollBar(JScrollBar.HORIZONTAL,w,50,0,w+50);
    scrW.addAdjustmentListener(this);
    add(scrW,"South");
    setVisible(true);
    add(content = new MyComponent(w,h));
    pack();
}
```

"ScalingFun.java"

Scrollbar





Beispiel – Scrollbar

```
    public void adjustmentValueChanged(AdjustmentEvent e) {
        Adjustable s = e.getAdjustable();
        int value = e.getValue();
        if (s == scrH) height = value;
        else width = value;
        revalidate();
        repaint();
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new ScalingFun());
    }
} // end of ScalingFun
```

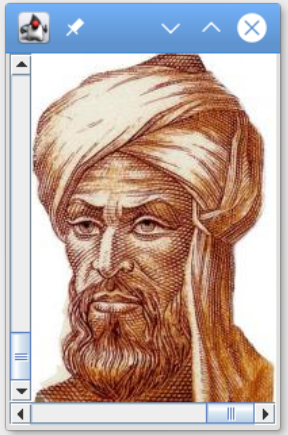
"ScalingFun.java"

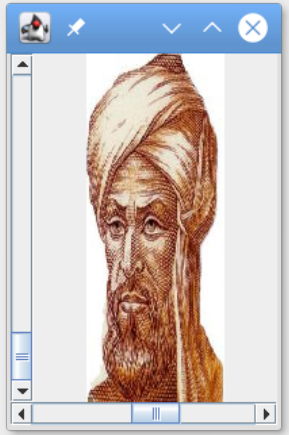
Erläuterungen

- Ein `JScrollBar`-Objekt erzeugt `AdjustmentEvent`-Ereignisse.
- Entsprechende Listener-Objekte müssen das Interface `AdjustmentListener` implementieren.
- Dieses verlangt die Implementierung einer Methode `void adjustmentValueChanged(AdjustmentEvent e);`
- Der Konstruktor legt zwei `JScrollBar`-Objekte an, eines horizontal, eines vertikal.

  Dafür gibt es in der Klasse `JScrollBar` die `int`-Konstanten `HORIZONTAL` und `VERTICAL`.

Scrollbar





Erläuterungen

- Ein `JScrollBar`-Objekt erzeugt `AdjustmentEvent`-Ereignisse.
- Entsprechende Listener-Objekte müssen das Interface `AdjustmentListener` implementieren.
- Dieses verlangt die Implementierung einer Methode `void adjustmentValueChanged(AdjustmentEvent e);`
- Der Konstruktor legt zwei `JScrollBar`-Objekte an, eines horizontal, eines vertikal.

  Dafür gibt es in der Klasse `JScrollBar` die `int`-Konstanten `HORIZONTAL` und `VERTICAL`.

Erläuterungen

- Der Konstruktor `JScrollBar(int dir,int init, int slide, int min, int max);` erzeugt eine JScrollBar der Ausrichtung `dir` mit Anfangsstellung `init`, Breite des Schiebers `slide`, minimalem Wert `min` und maximalem Wert `max`.

  Aufgrund der Breite des Schiebers ist der wirkliche Maximalwert `max - slide`.
- `void addAdjustmentListener(AdjustmentListener adj);` registriert das `AdjustmentListener`-Objekt als Listener für die `AdjustmentEvent`-Objekte der Scrollbars.

Erläuterungen

- Der Konstruktor `JScrollBar(int dir,int init, int slide, int min, int max);` erzeugt eine JScrollBar der Ausrichtung `dir` mit Anfangsstellung `init`, Breite des Schiebers `slide`, minimalem Wert `min` und maximalem Wert `max`.

  Aufgrund der Breite des Schiebers ist der wirkliche Maximalwert `max - slide`.
- `void addAdjustmentListener(AdjustmentListener adj);` registriert das `AdjustmentListener`-Objekt als Listener für die `AdjustmentEvent`-Objekte der Scrollbars.

Erläuterungen

- Um `AdjustmentEvent`-Objekte behandeln zu können, implementieren wir die Methode `AdjustmentValueChanged(AdjustmentEvent e);`
- Jedes `AdjustmentEvent`-Objekt verfügt über die Objekt-Methoden:

```
public AdjustmentListener getAdjustable();
public int getValue();
```

  ...mit denen das auslösende Objekt sowie der eingestellte `int`-Wert abgefragt werden kann.

Layouts

Bleibt, das Geheimnis um Layout und West bzw. South zu lüften...

- Jeder Container, in den man weitere Komponenten schachteln möchte, muss über eine Vorschrift verfügen, wie die Komponenten anzuordnen sind.
- Diese Vorschrift heißt Layout.

  Zur Festlegung des Layouts stellt Java das Interface LayoutManager zur Verfügung sowie nützliche implementierende Klassen...

Erläuterungen

- Um AdjustmentEvent-Objekte behandeln zu können, implementieren wir die Methode
  `AdjustmentValueChanged(AdjustmentEvent e);`
- Jedes AdjustmentEvent-Objekt verfügt über die Objekt-Methoden:

```
public AdjustmentListener getAdjustable();
public int getValue();
```

  ...mit denen das auslösende Objekt sowie der eingestellte int-Wert abgefragt werden kann.

Layouts

Bleibt, das Geheimnis um Layout und West bzw. South zu lüften...

- Jeder Container, in den man weitere Komponenten schachteln möchte, muss über eine Vorschrift verfügen, wie die Komponenten anzuordnen sind.
- Diese Vorschrift heißt Layout.

  Zur Festlegung des Layouts stellt Java das Interface LayoutManager zur Verfügung sowie nützliche implementierende Klassen...

Layouts

- Eine davon ist das BorderLayout.
- Mithilfe der String-Argumente:

```
BorderLayout.NORTH = "North",
BorderLayout.SOUTH = "South",
BorderLayout.WEST = "West",
BorderLayout.EAST = "East", und
BorderLayout.CENTER = "Center"
```

  kann man genau eine Komponente am bezeichneten Rand bzw. der Mitte positionieren.

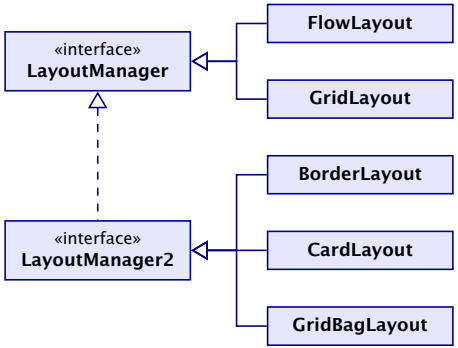
Layouts

- Eine davon ist das `BorderLayout`.
- Mithilfe der `String`-Argumente:

```
BorderLayout.NORTH = "North",
BorderLayout.SOUTH = "South",
BorderLayout.WEST = "West",
BorderLayout.EAST = "East", und
BorderLayout.CENTER = "Center"
```

kann man genau eine Komponente am bezeichneten Rand bzw. der Mitte positionieren.

Einige Layoutmanager



Layout Manager

FlowLayout: Komponenten werden von links nach rechts zeilenweise abgelegt; passt eine Komponente nicht mehr in eine Zeile, rückt sie in die nächste.

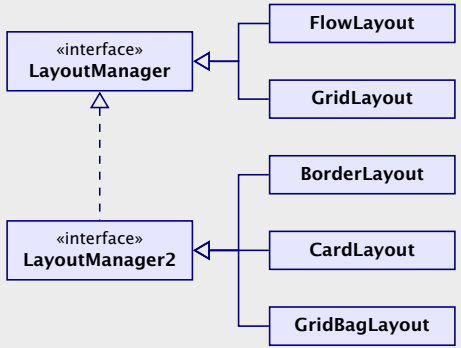
BorderLayout: Die Fläche wird in die fünf Regionen North, South, West, East und Center aufgeteilt, die jeweils von einer Komponente eingenommen werden können.

CardLayout: Die Komponenten werden wie in einem Karten-Stapel abgelegt. Der Stapel ermöglicht sowohl den Durchgang in einer festen Reihenfolge wie den Zugriff auf spezielle Elemente.

GridLayout: Die Komponenten werden in einem Gitter mit gegebener Zeilen- und Spalten-Anzahl abgelegt.

GridBagLayout: Wie `GridLayout`, nur flexibler, indem einzelne Komponenten auch mehrere Felder des Gitters belegen können.

Einige Layoutmanager



Ereignisse

- Komponenten erzeugen Ereignisse;
- Listener-Objekte werden an Komponenten für Ereignis-Klassen registriert;
- Ereignisse werden entsprechend ihrer Herkunft an Listener-Objekte weitergereicht.

Layout Manager

FlowLayout: Komponenten werden von links nach rechts zeilenweise abgelegt; passt eine Komponente nicht mehr in eine Zeile, rückt sie in die nächste.

BorderLayout: Die Fläche wird in die fünf Regionen North, South, West, East und Center aufgeteilt, die jeweils von einer Komponente eingenommen werden können.

CardLayout: Die Komponenten werden wie in einem Karten-Stapel abgelegt. Der Stapel ermöglicht sowohl den Durchgang in einer festen Reihenfolge wie den Zugriff auf spezielle Elemente.

GridLayout: Die Komponenten werden in einem Gitter mit gegebener Zeilen- und Spalten-Anzahl abgelegt.

GridBagLayout: Wie `GridLayout`, nur flexibler, indem einzelne Komponenten auch mehrere Felder des Gitters belegen können.

Ereignisse

- Jedes `AWTEvent`-Objekt verfügt über eine Quelle, d.h. eine Komponente, die dieses Ereignis erzeugte. `public Object getSource()` (der Klasse `java.util.EventObject`) liefert dieses Objekt.
- Gibt es verschiedene Klassen von Komponenten, die Ereignisse der gleichen Klasse erzeugen können, werden diese mit einem geeigneten Interface zusammengefasst.

Beispiele:

Ereignisklasse	*Interface*	*Objektmethode*
ItemEvent	ItemSelectable	getItemSelectable()
AdjustmentEvent	Adjustable	getAdjustable()

Ereignisse

- Komponenten erzeugen Ereignisse;
- Listener-Objekte werden an Komponenten für Ereignis-Klassen registriert;
- Ereignisse werden entsprechend ihrer Herkunft an Listener-Objekte weitergereicht.

Ereignisse

- Eine Komponente kann Ereignisse verschiedener `AWTEvent`-Klassen erzeugen.
- Für jede dieser Klassen können getrennt Listener-Objekte registriert werden...
- Man unterscheidet zwei Sorten von Ereignissen:
  1. semantische Ereignis-Klassen — wie `ActionEvent` oder `AdjustmentEvent`;
  2. low-level Ereignis-Klassen — wie `WindowEvent` oder `MouseEvent`.

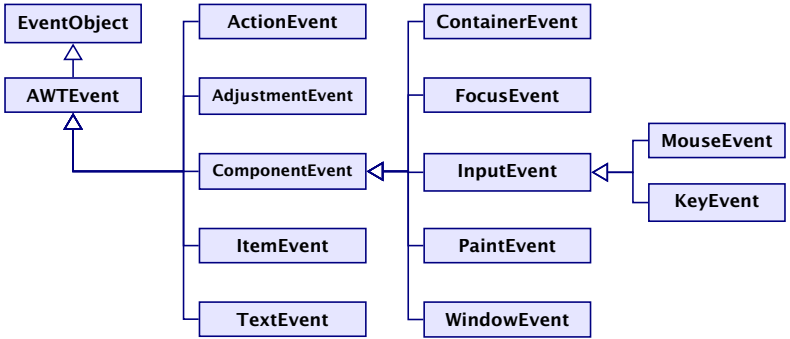
Ereignisse

- Jedes `AWTEvent`-Objekt verfügt über eine Quelle, d.h. eine Komponente, die dieses Ereignis erzeugte. `public Object getSource()` (der Klasse `java.util.EventObject`) liefert dieses Objekt.
- Gibt es verschiedene Klassen von Komponenten, die Ereignisse der gleichen Klasse erzeugen können, werden diese mit einem geeigneten Interface zusammengefasst.

Beispiele:

Ereignisklasse	*Interface*	*Objektmethode*
ItemEvent	ItemSelectable	getItemSelectable()
AdjustmentEvent	Adjustable	getAdjustable()

Überblick – Eventklassen



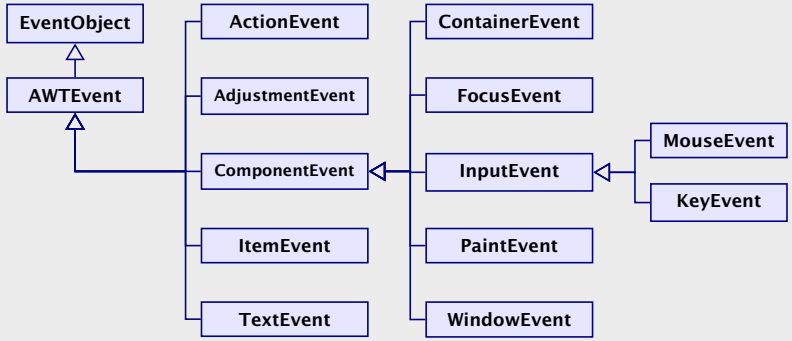
Ereignisse

- Eine Komponente kann Ereignisse verschiedener `AWTEvent`-Klassen erzeugen.
- Für jede dieser Klassen können getrennt Listener-Objekte registriert werden...
- Man unterscheidet zwei Sorten von Ereignissen:
  1. semantische Ereignis-Klassen — wie `ActionEvent` oder `AdjustmentEvent`;
  2. low-level Ereignis-Klassen — wie `WindowEvent` oder `MouseEvent`.

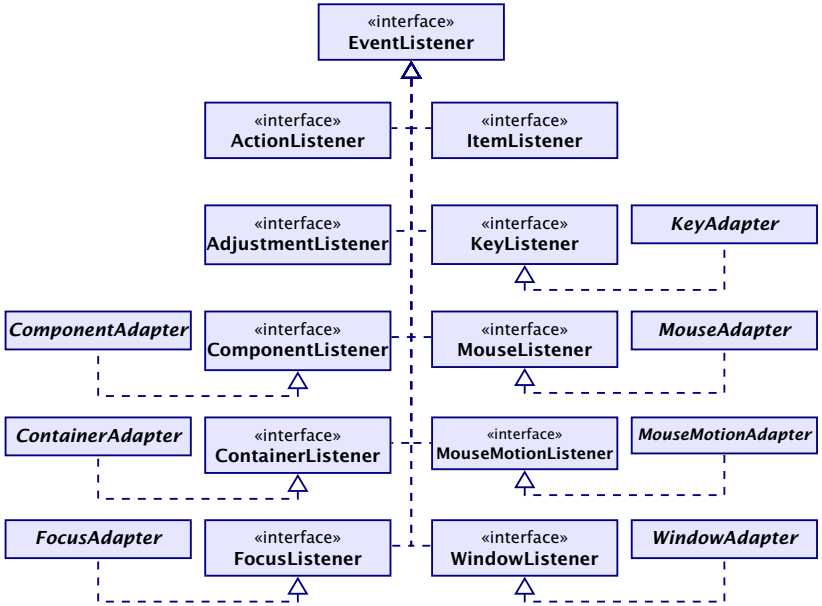
Listeners

- Zu jeder Klasse von Ereignissen gehört ein Interface, das die zuständigen Listener-Objekte implementieren müssen.
- Manche Interfaces verlangen die Implementierung mehrerer Methoden.
- In diesem Fall stellt Java Adapter-Klassen zur Verfügung.
- Die Adapterklasse zu einem Interface implementiert sämtliche geforderten Methoden auf triviale Weise ;-)
- In einer Unterklasse der Adapter-Klasse kann man sich darum darauf beschränken, nur diejenigen Methoden zu implementieren, auf die man Wert legt.

Überblick – Eventklassen



Überblick – Eventklassen



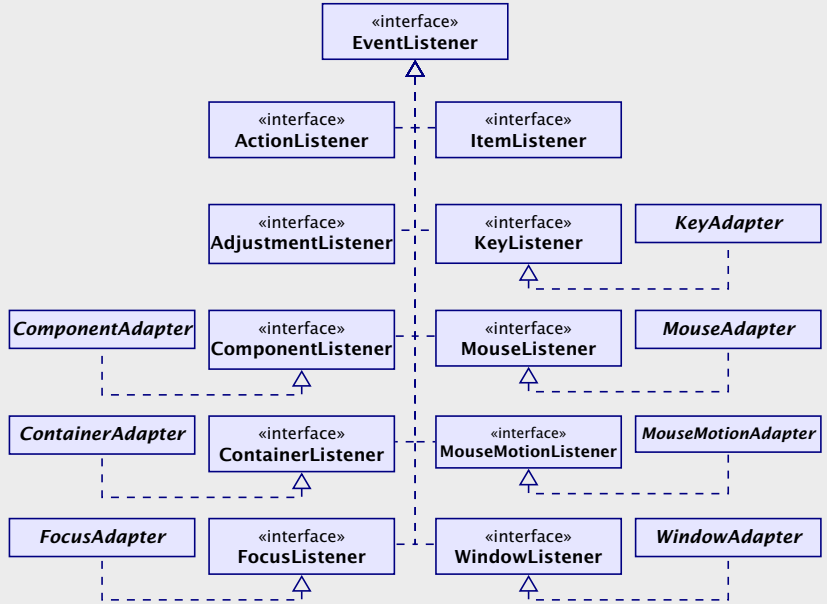
Listeners

- Zu jeder Klasse von Ereignissen gehört ein Interface, das die zuständigen Listener-Objekte implementieren müssen.
- Manche Interfaces verlangen die Implementierung mehrerer Methoden.
- In diesem Fall stellt Java Adapter-Klassen zur Verfügung.
- Die Adapterklasse zu einem Interface implementiert sämtliche geforderten Methoden auf triviale Weise ;-)
- In einer Unterklasse der Adapter-Klasse kann man sich darum darauf beschränken, nur diejenigen Methoden zu implementieren, auf die man Wert legt.

Beispiel – Ein MouseListener

- Das Interface `MouseListener` verlangt die Implementierung der Methoden:
  - `void mousePressed(MouseEvent e);`
  - `void mouseReleased(MouseEvent e);`
  - `void mouseEntered(MouseEvent e);`
  - `void mouseExited(MouseEvent e);`
  - `void mouseClicked(MouseEvent e);`
- Diese Methoden werden bei den entsprechenen Maus-Ereignissen der Komponente aufgerufen.
- Unser Beispielprogramm soll bei jedem Maus-Klick eine kleine Kreisfläche malen...

Überblick – Eventklassen



Beispiel – Ein MouseListener

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3
4  public class Mouse extends JFrame {
5      private Image buffer;
6      private JComponent comp;
7      private Graphics gBuff;
8
9      Mouse() {
10         setSize(500,500);
11         setVisible(true);
12         buffer = createImage(500,500);
13         gBuff  = buffer.getGraphics();
14         gBuff.setColor(Color.orange);
15         gBuff.fillRect(0,0,500,500);
16         comp = new MyComponent();
17         comp.addMouseListener(new MyMouseListener());
18         add(comp);
19     }
```

"Mouse.java"

Beispiel – Ein MouseListener

- Das Interface `MouseListener` verlangt die Implementierung der Methoden:
  - `void mousePressed(MouseEvent e);`
  - `void mouseReleased(MouseEvent e);`
  - `void mouseEntered(MouseEvent e);`
  - `void mouseExited(MouseEvent e);`
  - `void mouseClicked(MouseEvent e);`
- Diese Methoden werden bei den entsprechenen Maus-Ereignissen der Komponente aufgerufen.
- Unser Beispielprogramm soll bei jedem Maus-Klick eine kleine Kreisfläche malen...

Beispiel - Ein MouseListener

```
20      class MyMouseListener extends MouseAdapter {
21          public void mouseClicked(MouseEvent e) {
22              int x = e.getX(); int y = e.getY();
23              System.out.println("x:"+x+",y:"+y);
24              gBuff.setColor(Color.blue);
25              gBuff.fillOval(x-5,y-5,10,10);
26              repaint();
27          }
28      }
29      class MyComponent extends JComponent {
30          public void paintComponent(Graphics page) {
31              page.drawImage(buffer,0,0,500,500,null);
32          }
33      }
34      public static void main(String args[]) {
35          SwingUtilities.invokeLater(() -> new Mouse());
36      }
37  }
```

"Mouse.java"

Beispiel - Ein MouseListener

```
1  import javax.swing.*;
2  import java.awt.*; import java.awt.event.*;
3
4  public class Mouse extends JFrame {
5      private Image buffer;
6      private JComponent comp;
7      private Graphics gBuff;
8
9      Mouse() {
10         setSize(500,500);
11         setVisible(true);
12         buffer = createImage(500,500);
13         gBuff  = buffer.getGraphics();
14         gBuff.setColor(Color.orange);
15         gBuff.fillRect(0,0,500,500);
16         comp = new MyComponent();
17         comp.addMouseListener(new MyMouseListener());
18         add(comp);
19     }
```

"Mouse.java"

Erläuterungen

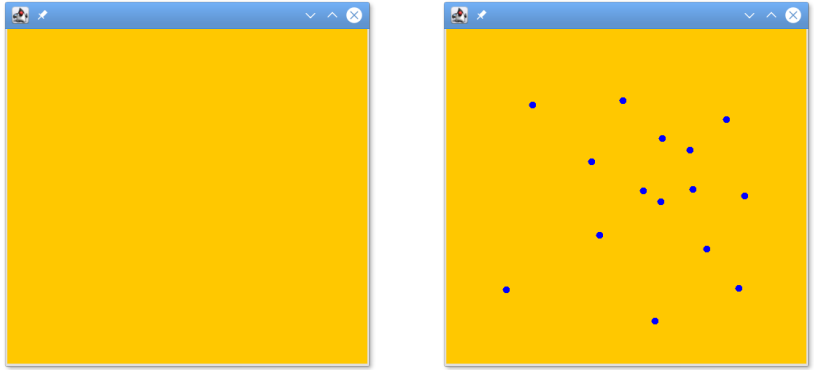
- Wir wollen nur die Methode `mouseClicked()` implementieren. Darum definieren wir unsere `MouseListener`-Klasse `MyMouseListener` als Unterklasse der Klasse `MouseAdapter`.
- Die `MouseEvent`-Methoden:
  `public int getX();` `public int getY();`
  liefern die Koordinaten, an denen der Mouse-Klick erfolgte...
- an dieser Stelle malen wir einen gefüllten Kreis in den Puffer.
- Dann rufen wir die Methode `repaint()` auf, um die Änderung sichtbar zu machen...

Beispiel – Ein MouseListener

```
20      class MyMouseListener extends MouseAdapter {
21          public void mouseClicked(MouseEvent e) {
22              int x = e.getX(); int y = e.getY();
23              System.out.println("x:"+x+",y:"+y);
24              gBuff.setColor(Color.blue);
25              gBuff.fillOval(x-5,y-5,10,10);
26              repaint();
27          }
28      }
29      class MyComponent extends JComponent {
30          public void paintComponent(Graphics page) {
31              page.drawImage(buffer,0,0,500,500,null);
32          }
33      }
34      public static void main(String args[]) {
35          SwingUtilities.invokeLater(() -> new Mouse());
36      }
37  }
```

"Mouse.java"

Beispiel – MouseListener



Erläuterungen

- Wir wollen nur die Methode `mouseClicked()` implementieren. Darum definieren wir unsere `MouseListener`-Klasse `MyMouseListener` als Unterklasse der Klasse `MouseAdapter`.
- Die `MouseEvent`-Methoden:
  ```
  public int getX();    public int getY();
  ```
  liefern die Koordinaten, an denen der Mouse-Klick erfolgte...
- an dieser Stelle malen wir einen gefüllten Kreis in den Puffer.
- Dann rufen wir die Methode `repaint()` auf, um die Änderung sichtbar zu machen...

18 Tic-Tac-Toe

Regeln:

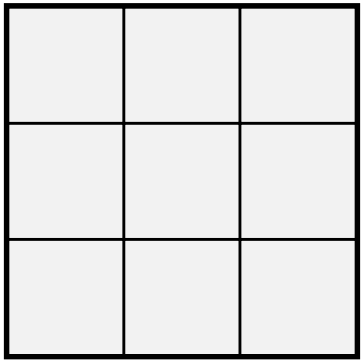
- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

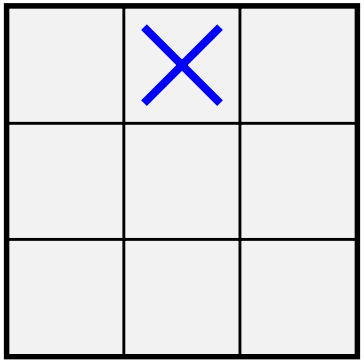


18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

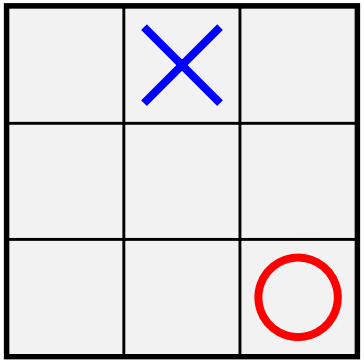


18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

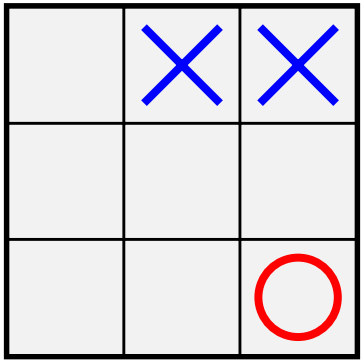


18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

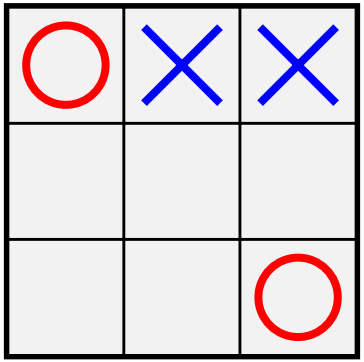


18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

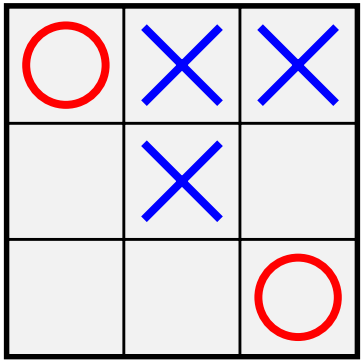


18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

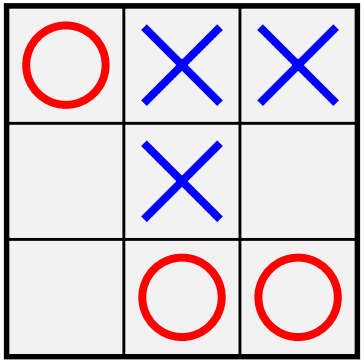


18 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel

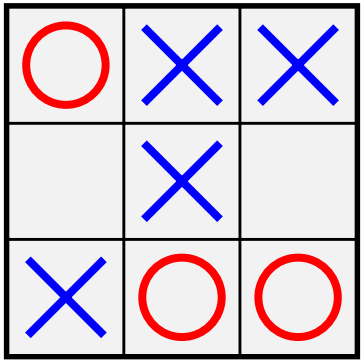


18 Tic-Tac-Toe

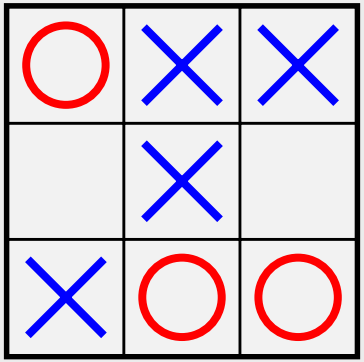
Regeln:

- Zwei Personen setzen abwechselnd Steine auf ein (3×3)-Spielfeld.
- Wer zuerst drei Steine in einer Reihe erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel



Beispiel



Analyse

. . . offenbar hat die anziehende Partei gewonnen.

Fragen

- Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- Wie implementiert man ein Tic-Tac-Toe-Programm, das
  - ...möglichst oft gewinnt?
  - ...eine ansprechende Oberfläche bietet?

Hintergrund — Zwei-Personen-Nullsummenspiele

Tic-Tac-Toe ist ein endliches Zwei-Personen-Nullsummen-Spiel, mit perfekter Information. Das heißt:

- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von Spiel-Konfigurationen.
- Die Spieler ziehen abwechselnd. Ein Zug wechselt die Konfiguration, bis eine Endkonfiguration erreicht ist.
- Jede Endkonfiguration ist mit einem Gewinn aus $\mathbb{R}$ bewertet.
- Person 0 gewinnt, wenn Endkonfiguration mit negativem Gewinn erreicht wird; sonst gewinnt Person 1.

Analyse

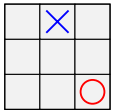
. . . offenbar hat die anziehende Partei gewonnen.

Fragen

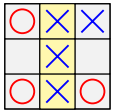
- Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- Wie implementiert man ein Tic-Tac-Toe-Programm, das
  - ...möglichst oft gewinnt?
  - ...eine ansprechende Oberfläche bietet?

...im Beispiel

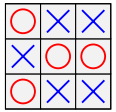
Konfiguration:



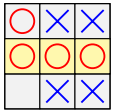
Endkonfigurationen:



Gewinn -1



Gewinn 0



Gewinn +1

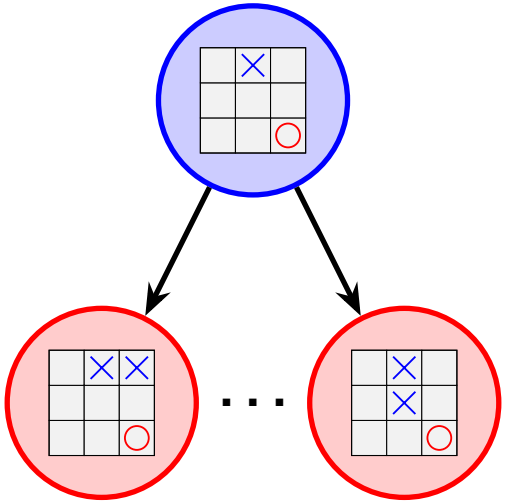
Hintergrund — Zwei-Personen-Nullsummenspiele

Tic-Tac-Toe ist ein endliches Zwei-Personen-Nullsummen-Spiel, mit perfekter Information. Das heißt:

- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von Spiel-Konfigurationen.
- Die Spieler ziehen abwechselnd. Ein Zug wechselt die Konfiguration, bis eine Endkonfiguration erreicht ist.
- Jede Endkonfiguration ist mit einem Gewinn aus $\mathbb{R}$ bewertet.
- Person 0 gewinnt, wenn Endkonfiguration mit negativem Gewinn erreicht wird; sonst gewinnt Person 1.

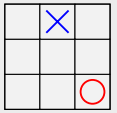
...im Beispiel

Spielzug:

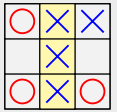


...im Beispiel

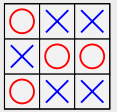
Konfiguration:



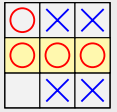
Endkonfigurationen:



Gewinn -1



Gewinn 0



Gewinn +1

Spielbaum

Ein Spielbaum wird folgendermassen (rekursiv) konstruiert:

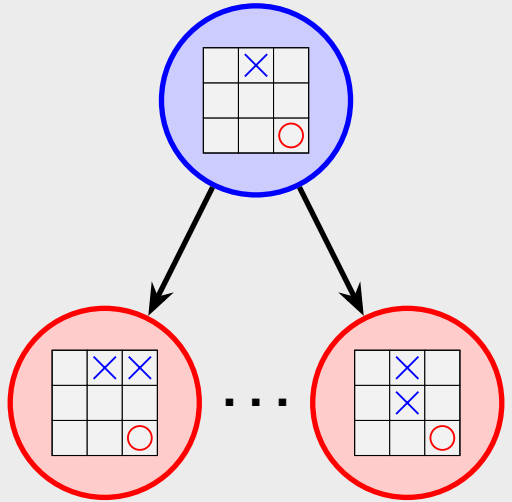
- gegeben ein Knoten v, der eine Spielkonfiguration repräsentiert
- für jede mögliche Nachfolgekonfiguration erzeugen wir einen Kindknoten, den wir mit v verbinden;
- dann starten wir den Prozess rekursiv für alle Kindknoten.

Eigenschaften:

- jeder Knoten repräsentiert eine Konfiguration; allerdings kann dieselbe Konfiguration sehr oft vorkommen
- Blattknoten repräsentieren Endkonfigurationen
- Kanten repräsentieren Spielzüge
- jedes Spiel ist ein Pfad von der Wurzel zu einem Blatt

...im Beispiel

Spielzug:



Spielbaum

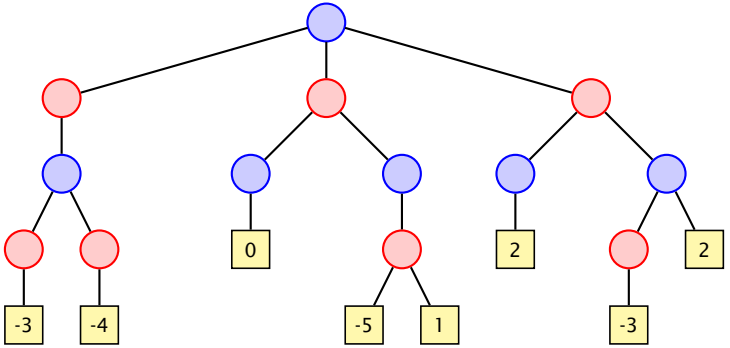
Ein Spielbaum wird folgendermassen (rekursiv) konstruiert:

- gegeben ein Knoten v, der eine Spielkonfiguration repräsentiert
- für jede mögliche Nachfolgekonfiguration erzeugen wir einen Kindknoten, den wir mit v verbinden;
- dann starten wir den Prozess rekursiv für alle Kindknoten.

Eigenschaften:

- jeder Knoten repräsentiert eine Konfiguration; allerdings kann dieselbe Konfiguration sehr oft vorkommen
- Blattknoten repräsentieren Endkonfigurationen
- Kanten repräsentieren Spielzüge
- jedes Spiel ist ein Pfad von der Wurzel zu einem Blatt

Beispiel — Spielbaum

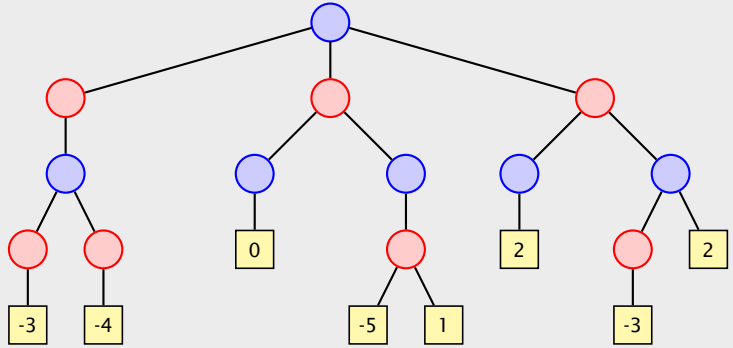


Spielbaum

Fragen:

- Wie finden wir uns (z.B. als blaue Person) im Spielbaum zurecht?
- Was müssen wir tun, um sicher ein negatives Blatt zu erreichen?

Beispiel — Spielbaum



Spielbaum

Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Spielbaum

Fragen:

- Wie finden wir uns (z.B. als blaue Person) im Spielbaum zurecht?
- Was müssen wir tun, um sicher ein negatives Blatt zu erreichen?

Spielbaum

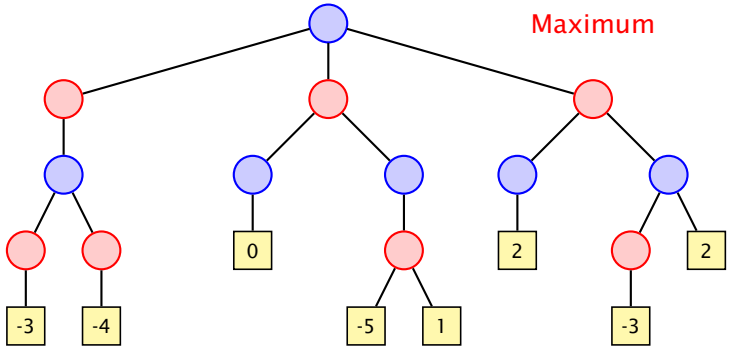
Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum



Spielbaum

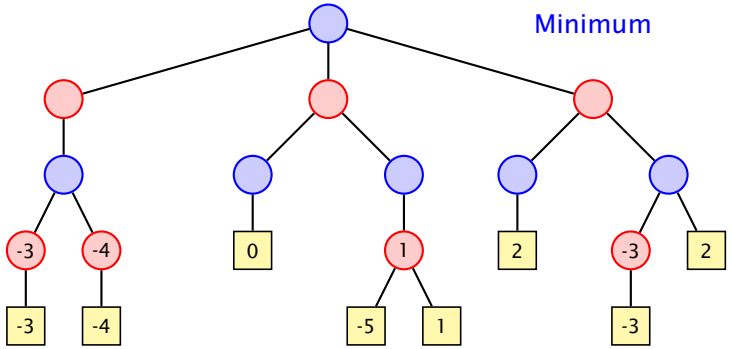
Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum



Spielbaum

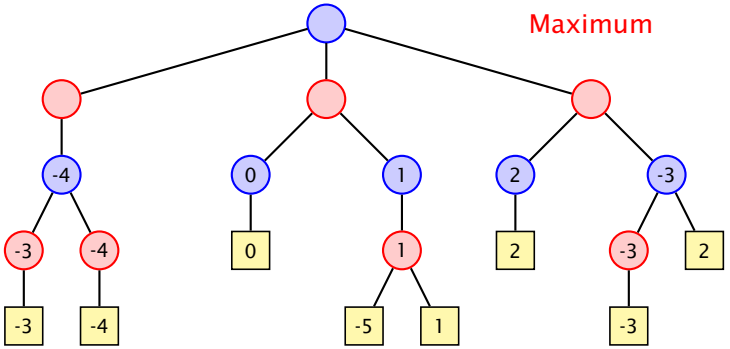
Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum



Spielbaum

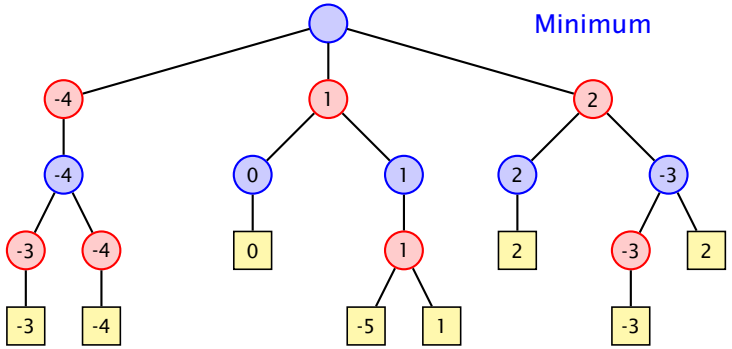
Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum



Spielbaum

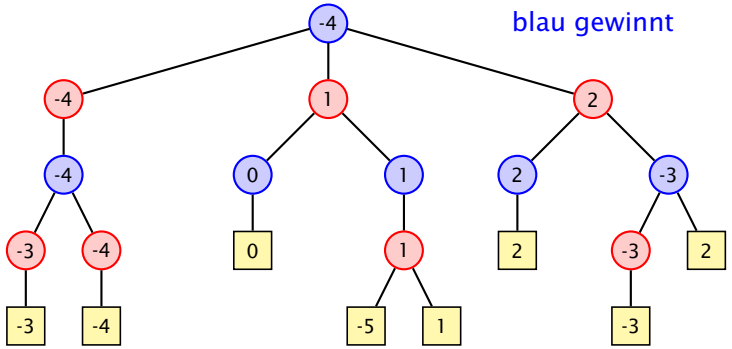
Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum



Spielbaum

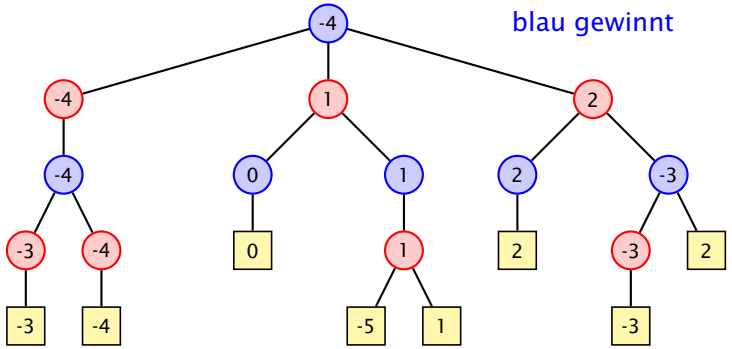
Idee:

- Wir ermitteln für jede Konfiguration den jeweils besten zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

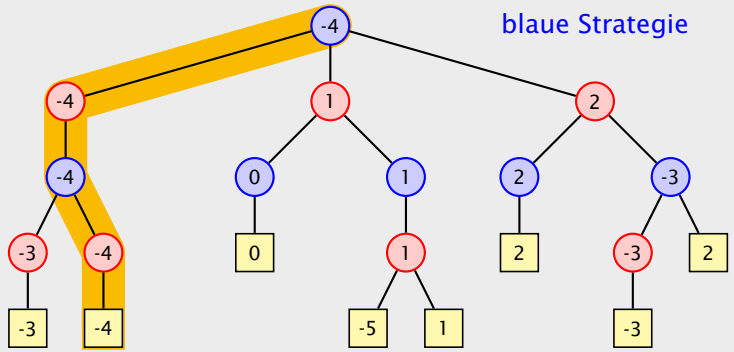
Fall 1 Die Konfiguration ist blau: wir sind am Zug. Wir können garantiert das Minimum der Gewinne der Söhne erzielen.

Fall 2 Die Konfiguration ist rot: der Gegner ist am Zug. Er kann garantiert das Maximum der Gewinne der Söhne erzielen.

Beispiel — Spielbaum



Beispiel — Spielbaum



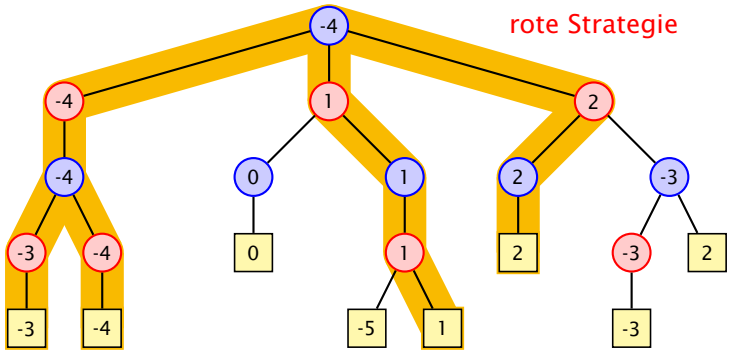
Strategien

Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine optimale Strategie ist eine, deren Anwendung garantiert zu einer Endkonfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...

Beispiel — Spielbaum



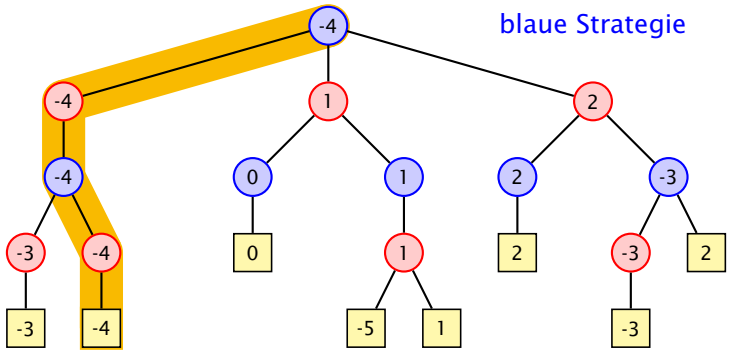
Strategien

Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine optimale Strategie ist eine, deren Anwendung garantiert zu einer Endkonfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...

Beispiel — Spielbaum



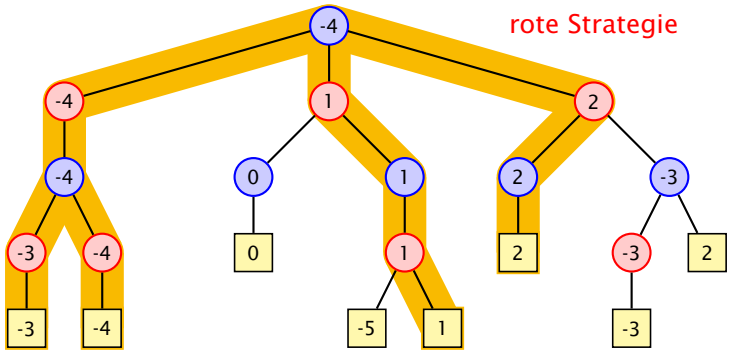
Strategien

Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine optimale Strategie ist eine, deren Anwendung garantiert zu einer Endkonfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...

Beispiel — Spielbaum



Strategien

Eine Strategie ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine optimale Strategie ist eine, deren Anwendung garantiert zu einer Endkonfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine akzeptable Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...

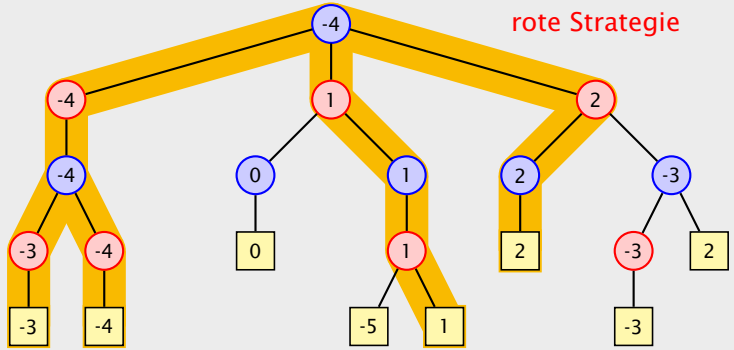
Struktur

GameTreeNode
- pos : Position - type : int - value : int - child : GameTreeNode[]
+ GameTreeNode (Position p) + bestMove () : int

Position
- playerToMove : int - arena : int[]
+ Position () + Position (Position p) + won () : int + getMoves () : Iterator<Integer> + makeMove (int m) + movePossible (int m) : boolean + getPlToMv () : int

Game
- g : GameTreeNode - p : Position
+ makeBestMove () + makePlayerMove () + movePossible () : boolean + finished () : boolean

Beispiel — Spielbaum



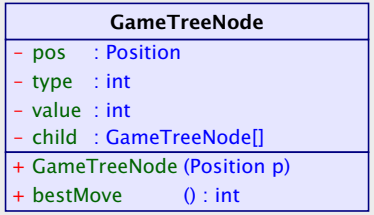
Implementierung – SpielbaumA

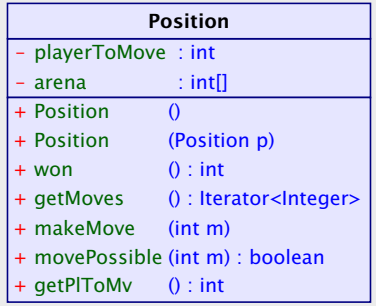
```java
import java.util.*;
public class GameTreeNode implements PlayConstants {
    static public int nodeCount = 0;

    private int value;
    private int type;
    private int bestMove = -1;
    private Position pos;
    private GameTreeNode[] child = new GameTreeNode[9];

    public int bestMove() {
        return bestMove;
    }
```

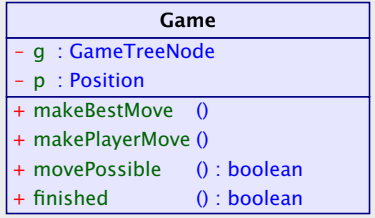
"GameTreeNodeA.java"

- das interface `PlayConstants` definiert die Konstanten `MIN` = -1, `NONE` = 0, `MAX` = 1;

Struktur







Implementierung – SpielbaumA

```
14      public GameTreeNode(Position p) { nodeCount++;
15          pos = p; type = p.playerToMove;
16          // hab ich schon verloren?
17          if (p.won() == -type) { value = -type; return; }
18          // no more moves --> no winner
19          Iterator<Integer> moves = p.getMoves();
20          if (!moves.hasNext()) { value = NONE; return; }
21          value = -2*type;
22          while (moves.hasNext()) {
23              int m = moves.next();
24              child[m] = new GameTreeNode(p.makeMove(m));
25              if (type == MIN && child[m].value < value ||
26                  type == MAX && child[m].value > value) {
27                  value = child[m].value;
28                  bestMove = m;
29  }   }   }   }
```

"GameTreeNodeA.java"

Implementierung – SpielbaumA

```
1  import java.util.*;
2  public class GameTreeNode implements PlayConstants {
3      static public int nodeCount = 0;
4
5      private int value;
6      private int type;
7      private int bestMove = -1;
8      private Position pos;
9      private GameTreeNode[] child = new GameTreeNode[9];
10
11     public int bestMove() {
12         return bestMove;
13     }
```

"GameTreeNodeA.java"

- das interface PlayConstants definiert die Konstanten MIN = -1, NONE = 0, MAX = 1;

Implementierung – SpielbaumA

Die einzigen TicTacToe-spezifischen Informationen in der Klasse GameTreeNode sind

- die Größe des Arrays child; wir wissen, dass wir höchstens 9 Züge machen können
- wir kennen die Gewinnwerte:

  MIN gewinnt : value = -1
  unentschieden : value = 0
  MAX gewinnt : value = +1

  deswegen könne wir z.B. value mit -2*type initialisieren.

Die anderen Regeln werden in die Klasse Position ausgelagert.

Implementierung – SpielbaumA

```
14      public GameTreeNode(Position p) { nodeCount++;
15          pos = p; type = p.playerToMove;
16          // hab ich schon verloren?
17          if (p.won() == -type) { value = -type; return; }
18          // no more moves --> no winner
19          Iterator<Integer> moves = p.getMoves();
20          if (!moves.hasNext()) { value = NONE; return; }
21          value = -2*type;
22          while (moves.hasNext()) {
23              int m = moves.next();
24              child[m] = new GameTreeNode(p.makeMove(m));
25              if (type == MIN && child[m].value < value ||
26                  type == MAX && child[m].value > value) {
27                  value = child[m].value;
28                  bestMove = m;
29  }   }   }   }
```

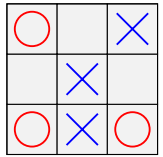
"GameTreeNodeA.java"

Klasse Position – Kodierung

Das Array `arena` enthält die Spielstellung z.B.:
`arena = {1,0,-1,0,-1,0,1,-1,1}` bedeutet:

0	1	2
3	4	5
6	7	8

Koordinaten



Konfiguration

1	0	-1
0	-1	0
1	-1	1

Kodierung

Implementierung – SpielbaumA

Die einzigen TicTacToe-spezifischen Informationen in der Klasse `GameTreeNode` sind

- die Größe des Arrays `child`; wir wissen, dass wir höchstens 9 Züge machen können
- wir kennen die Gewinnwerte:

  `MIN` gewinnt : `value` = -1
  unentschieden : `value` = 0
  `MAX` gewinnt : `value` = +1

  deswegen könne wir z.B. `value` mit `-2*type` initialisieren.

Die anderen Regeln werden in die Klasse `Position` ausgelagert.

Implementierung – Position

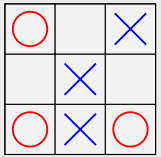
```
1  public class Position implements PlayConstants {
2      private int[] arena;
3      private int playerToMove = MIN;
4      public Position() { arena = new int[9]; }
5      public Position(Position p) {
6          arena = (int[]) p.arena.clone();
7          playerToMove = p.playerToMove;
8      }
9      public Position makeMove(int place) {
10         Position p = new Position(this);
11         p.arena[place] = playerToMove;
12         p.playerToMove = -playerToMove;
13         return p;
14     }
15     private boolean free(int place) {
16         return (arena[place] == NONE);
17     }
18     public boolean movePossible(int pl) {
19         return (getMoves().hasNext() && free(pl));
20     }
```

Klasse Position – Kodierung

Das Array `arena` enthält die Spielstellung z.B.:
`arena = {1,0,-1,0,-1,0,1,-1,1}` bedeutet:

0	1	2
3	4	5
6	7	8

Koordinaten



Konfiguration

1	0	-1
0	-1	0
1	-1	1

Kodierung

Implementierung – Position

```
21 private class PossibleMoves implements Iterator<Integer> {
22     private int next = 0;
23     public boolean hasNext() {
24         if (won() != 0) return false;
25         for (; next<9; next++)
26             if (free(next)) return true;
27         return false;
28     }
29     public Integer next() {
30         if (won() == 0)
31             for(; next<9; next++)
32                 if (free(next)) return next++;
33         throw new NoSuchElementException();
34 }   }
35 public Iterator<Integer> getMoves() {
36     return new PossibleMoves();
37 }
```

"Position.java"

Implementierung – Position

```
1  public class Position implements PlayConstants {
2      private int[] arena;
3      private int playerToMove = MIN;
4      public Position() { arena = new int[9]; }
5      public Position(Position p) {
6          arena = (int[]) p.arena.clone();
7          playerToMove = p.playerToMove;
8      }
9      public Position makeMove(int place) {
10         Position p = new Position(this);
11         p.arena[place] = playerToMove;
12         p.playerToMove = -playerToMove;
13         return p;
14     }
15     private boolean free(int place) {
16         return (arena[place] == NONE);
17     }
18     public boolean movePossible(int pl) {
19         return (getMoves().hasNext() && free(pl));
20     }
```

Klasse Game

Die Klasse Game sammelt notwendige Datenstrukturen und Methoden zur Durchführung des Spiels:

```
1  public class Game implements PlayConstants, Model {
2      private Position     p;
3      private GameTreeNode g;
4      private View view;
5
6      Game(View v) {
7          view = v;
8          p = new Position();
9      }
10     private void initTree() {
11         g.nodeCount = 0;
12         g = new GameTreeNode(p);
13         System.out.println("generate tree... (" +
14                            g.nodeCount + " nodes)");
15     }
```

"Game.java"

Implementierung – Position

```
21 private class PossibleMoves implements Iterator<Integer> {
22     private int next = 0;
23     public boolean hasNext() {
24         if (won() != 0) return false;
25         for (; next<9; next++)
26             if (free(next)) return true;
27         return false;
28     }
29     public Integer next() {
30         if (won() == 0)
31             for(; next<9; next++)
32                 if (free(next)) return next++;
33         throw new NoSuchElementException();
34 }   }
35 public Iterator<Integer> getMoves() {
36     return new PossibleMoves();
37 }
```

"Position.java"

Klasse Game

```
17      private void makeMove(int place) {
18          view.put(place,p.getPlToMv());
19          p = p.makeMove(place);
20          if (finished())
21              view.showWinner(p.won());
22      }
23      public void makeBestMove() {
24          initTree();
25          makeMove(g.bestMove());
26      }
27      public void makePlayerMove(int place) {
28          makeMove(place);
29          if (!finished()) {
30              makeBestMove();
31          }
32      }
```

"Game.java"

Klasse Game

Die Klasse Game sammelt notwendige Datenstrukturen und Methoden zur Durchführung des Spiels:

```
1  public class Game implements PlayConstants, Model {
2      private Position     p;
3      private GameTreeNode g;
4      private View view;
5
6      Game(View v) {
7          view = v;
8          p = new Position();
9      }
10     private void initTree() {
11         g.nodeCount = 0;
12         g = new GameTreeNode(p);
13         System.out.println("generate tree... (" +
14                                 g.nodeCount + " nodes)");
15     }
```

"Game.java"

Klasse Game

```
33      public boolean movePossible(int place) {
34          return p.movePossible(place);
35      }
36      public boolean finished() {
37          return !p.getMoves().hasNext();
38      }
39      public static void main(String[] args) {
40          Game game = new Game(new DummyView());
41          for (int i = 0; i < 9; ++i) {
42              if (!game.finished()) {
43                  game.makeBestMove();
44                  System.out.println(game.p);
45              }
46              else System.out.println("no more moves");
47  }   }   }
```

"Game.java"

Klasse Game

```
17      private void makeMove(int place) {
18          view.put(place,p.getPlToMv());
19          p = p.makeMove(place);
20          if (finished())
21              view.showWinner(p.won());
22      }
23      public void makeBestMove() {
24          initTree();
25          makeMove(g.bestMove());
26      }
27      public void makePlayerMove(int place) {
28          makeMove(place);
29          if (!finished()) {
30              makeBestMove();
31          }
32      }
```

"Game.java"

Output – Variante A

```
generate tree... (549946 nodes)
x..
...
...
generate tree... (59705 nodes)
x..
.o.
...
generate tree... (7332 nodes)
xx.
.o.
...
generate tree... (935 nodes)
xxo
.o.
...
generate tree... (198 nodes)
xxo
.o.
x..
```

```
generate tree... (47 nodes)
xxo
oo.
x..
generate tree... (14 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Klasse Game

```
33      public boolean movePossible(int place) {
34          return p.movePossible(place);
35      }
36      public boolean finished() {
37          return !p.getMoves().hasNext();
38      }
39      public static void main(String[] args) {
40          Game game = new Game(new DummyView());
41          for (int i = 0; i < 9; ++i) {
42              if (!game.finished()) {
43                  game.makeBestMove();
44                  System.out.println(game.p);
45              }
46              else System.out.println("no more moves");
47  }   }   }
```

"Game.java"

Effizienz

Wie können wir das effizienter gestalten?

1. Den Spielbaum nur einmal berechnen, anstatt jedesmal neu.
   gewinnt nicht sehr viel...
2. Wenn wir z.B. als MaxPlayer schon einen Wert von 1 erreicht haben, brauchen wir nicht weiterzusuchen...

   Spielbaum ist dann unvollständig; Wiederverwendung schwierig...

$\Longrightarrow$ Baue keinen vollständigen Spielbaum; nur Wert und Zug an der Wurzel müssen korrekt sein.

Output – Variante A

```
generate tree... (549946 nodes)
x..
...
...
generate tree... (59705 nodes)
x..
.o.
...
generate tree... (7332 nodes)
xx.
.o.
...
generate tree... (935 nodes)
xxo
.o.
...
generate tree... (198 nodes)
xxo
.o.
x..
```

```
generate tree... (47 nodes)
xxo
oo.
x..
generate tree... (14 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Implementierung – SpielbaumB

```
14      public GameTreeNode(Position p) { nodeCount++;
15          pos = p; type = p.playerToMove;
16          // hab ich schon verloren?
17          if (p.won() == -type) { value = -type; return; }
18          // no more moves --> no winner
19          Iterator<Integer> moves = p.getMoves();
20          if (!moves.hasNext()) { value = NONE; return; }
21          value = -2*type;
22          while (moves.hasNext()) {
23              int m = moves.next();
24              child[m] = new GameTreeNode(p.makeMove(m));
25              if (type == MIN && child[m].value < value ||
26                  type == MAX && child[m].value > value) {
27                  value = child[m].value;
28                  bestMove = m;
29                  // we won; don't search further
30                  if (value == type) return;
31  }   }   }   }
```

"GameTreeNodeB.java"

Effizienz

Wie können wir das effizienter gestalten?

1. Den Spielbaum nur einmal berechnen, anstatt jedesmal neu.
   gewinnt nicht sehr viel...
2. Wenn wir z.B. als MaxPlayer schon einen Wert von 1 erreicht haben, brauchen wir nicht weiterzusuchen...

   Spielbaum ist dann unvollständig; Wiederverwendung schwierig...

$\Longrightarrow$ Baue keinen vollständigen Spielbaum; nur Wert und Zug an der Wurzel müssen korrekt sein.

Output – Variante B

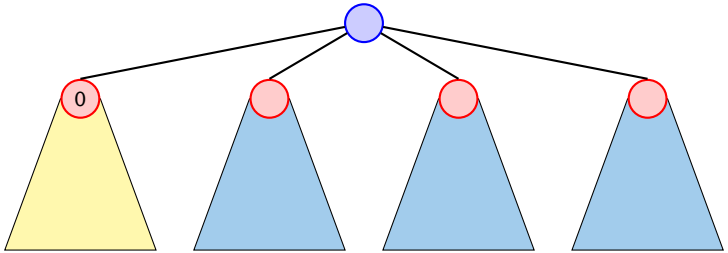
```
generate tree... (94978 nodes)
x..
...
...
generate tree... (3763 nodes)
x..
.o.
...
generate tree... (1924 nodes)
xx.
.o.
...
generate tree... (61 nodes)
xxo
.o.
...
generate tree... (50 nodes)
xxo
.o.
x..
generate tree... (17 nodes)
xxo
oo.
x..
generate tree... (10 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Implementierung – SpielbaumB

```
14      public GameTreeNode(Position p) { nodeCount++;
15          pos = p; type = p.playerToMove;
16          // hab ich schon verloren?
17          if (p.won() == -type) { value = -type; return; }
18          // no more moves --> no winner
19          Iterator<Integer> moves = p.getMoves();
20          if (!moves.hasNext()) { value = NONE; return; }
21          value = -2*type;
22          while (moves.hasNext()) {
23              int m = moves.next();
24              child[m] = new GameTreeNode(p.makeMove(m));
25              if (type == MIN && child[m].value < value ||
26                  type == MAX && child[m].value > value) {
27                  value = child[m].value;
28                  bestMove = m;
29                  // we won; don't search further
30                  if (value == type) return;
31  }   }   }   }
```

"GameTreeNodeB.java"

Effizienz – Alpha-Beta-Pruning



Ein Wert > 0 innerhalb der blauen Teilbäume kann nicht zu Wurzel gelangen (Wurzel ist MIN-Knoten). Deshalb kann ein MAX-Knoten innerhalb dieser Bäume abbrechen, wenn er einen Wert ≥ 0 erzielt hat.

Analog für MIN.

Output – Variante B

```
generate tree... (94978 nodes)
x..
...
...
generate tree... (3763 nodes)
x..
.o.
...
generate tree... (1924 nodes)
xx.
.o.
...
generate tree... (61 nodes)
xxo
.o.
...
generate tree... (50 nodes)
xxo
.o.
x..
```

```
generate tree... (17 nodes)
xxo
oo.
x..
generate tree... (10 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

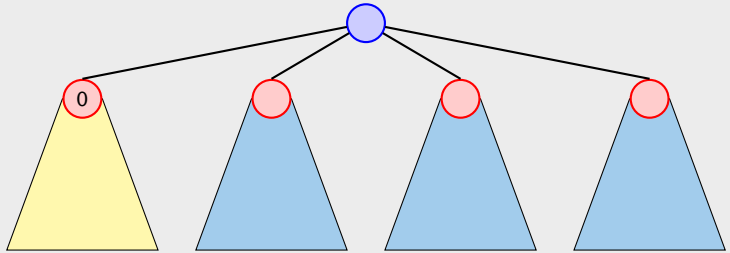
Implementierung – SpielbaumC

Änderungen am Konstruktor:

```
1     private GameTreeNode(Position p,
2                          int goalMin, int goalMax) {
3         nodeCount++;
4         pos = p; type = p.playerToMove;
5         if (p.won() == -type) { value = -type; return; }
6         Iterator<Integer> moves = p.getMoves();
7         if (!moves.hasNext()) { value = NONE; return; }
8
9         value = -2*type;
10        while (moves.hasNext()) {
11            int m = moves.next();
12            child[m] = new GameTreeNode(p.makeMove(m),
13                                        goalMin,goalMax);
14    // continued...
```

"GameTreeNodeC.java"

Effizienz – Alpha-Beta-Pruning



Ein Wert > 0 innerhalb der blauen Teilbäume kann nicht zu Wurzel gelangen (Wurzel ist MIN-Knoten). Deshalb kann ein MAX-Knoten innerhalb dieser Bäume abbrechen, wenn er einen Wert ≥ 0 erzielt hat.

Analog für MIN.

Implementierung – SpielbaumC

```
15          if (type == MIN && child[m].value < value ||
16              type == MAX && child[m].value > value) {
17              value = child[m].value;
18              bestMove = m;
19
20              // leave if goal is reached
21              if (type == MIN && value <= goalMin) return;
22              if (type == MAX && value >= goalMax) return;
23              // update goals
24              if (type == MIN) goalMax = value;
25              if (type == MAX) goalMin = value;
26      }  }  } // if, while, Konstruktor
27      public GameTreeNode(Position p) {
28          this(p, MIN, MAX);
29      }
30  }
```

"GameTreeNodeC.java"

Implementierung – SpielbaumC

Änderungen am Konstruktor:

```
1       private GameTreeNode(Position p,
2                            int goalMin, int goalMax) {
3           nodeCount++;
4           pos = p; type = p.playerToMove;
5           if (p.won() == -type) { value = -type; return; }
6           Iterator<Integer> moves = p.getMoves();
7           if (!moves.hasNext()) { value = NONE; return; }
8
9           value = -2*type;
10          while (moves.hasNext()) {
11              int m = moves.next();
12              child[m] = new GameTreeNode(p.makeMove(m),
13                                          goalMin,goalMax);
14  // continued...
```

"GameTreeNodeC.java"

Output – Variante C

```
generate tree... (18014 nodes)
x..
...
...
generate tree... (1957 nodes)
x..
.o.
...
generate tree... (764 nodes)
xx.
.o.
...
generate tree... (61 nodes)
xxo
.o.
...
generate tree... (50 nodes)
xxo
.o.
x..
generate tree... (17 nodes)
xxo
oo.
x..
generate tree... (10 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Implementierung – SpielbaumC

```
15            if (type == MIN && child[m].value < value ||
16                type == MAX && child[m].value > value) {
17                value = child[m].value;
18                bestMove = m;
19
20                // leave if goal is reached
21                if (type == MIN && value <= goalMin) return;
22                if (type == MAX && value >= goalMax) return;
23                // update goals
24                if (type == MIN) goalMax = value;
25                if (type == MAX) goalMin = value;
26        }  }  } // if, while, Konstruktor
27        public GameTreeNode(Position p) {
28            this(p, MIN, MAX);
29        }
30    }
```

"GameTreeNodeC.java"

Effizienz

Bis jetzt haben wir bei den Effizienzsteigerungen das eigentliche Spiel ignoriert.

- Wenn wir einen Zug haben, der sofort gewinnt, kennen wir den Wert des Knotens und den besten Zug.
- Falls das nicht zutrifft, aber der Gegner am Zug einen sofortigen Gewinn hätte, dann ist der beste Zug dieses zu verhindern. D.h. wir kennen den besten Zug, aber noch nicht den Wert des Knotens.

`int forcedWin(int player)` in der Klasse `Position` überprüft, ob `player` einen Zug mit sofortigem Gewinn hat.

- falls ja, gibt es diesen Zug zurück
- sonst gibt es `-1` zurück

Output – Variante C

```
generate tree... (18014 nodes)
x..
...
...
generate tree... (1957 nodes)
x..
.o.
...
generate tree... (764 nodes)
xx.
.o.
...
generate tree... (61 nodes)
xxo
.o.
...
generate tree... (50 nodes)
xxo
.o.
x..
```

```
generate tree... (17 nodes)
xxo
oo.
x..
generate tree... (10 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Implementierung – SpielbaumD

```
1     private GameTreeNode(Position p,
2                          int goalMin, int goalMax) {
3         nodeCount++; pos = p; type = p.getPlToMv();
4         if (p.won() == -type) { value = -type; return; }
5         Iterator<Integer> moves = p.getMoves();
6         if (!moves.hasNext()) { value = NONE; return; }
7         int m;
8         if ((m=p.forcedWin(type)) != -1) {
9             bestMove = m;
10            value = type;
11            return;
12        }
13        if ((m=p.forcedWin(-type)) != -1) {
14            bestMove = m;
15            child[m] = new GameTreeNode(p.makeMove(m),
16                                        goalMin, goalMax);
17            value = child[m].value;
18            return;
19        }
```

"GameTreeNodeD.java"

Effizienz

Bis jetzt haben wir bei den Effizienzsteigerungen das eigentliche Spiel ignoriert.

- Wenn wir einen Zug haben, der sofort gewinnt, kennen wir den Wert des Knotens und den besten Zug.
- Falls das nicht zutrifft, aber der Gegner am Zug einen sofortigen Gewinn hätte, dann ist der beste Zug dieses zu verhindern. D.h. wir kennen den besten Zug, aber noch nicht den Wert des Knotens.

`int forcedWin(int player)` in der Klasse `Position` überprüft, ob `player` einen Zug mit sofortigem Gewinn hat.

- falls ja, gibt es diesen Zug zurück
- sonst gibt es `-1` zurück

Implementierung – SpielbaumD

```
20        value = -2*type;
21        while (moves.hasNext()) {
22           m = moves.next();
23           child[m] = new GameTreeNode(p.makeMove(m),
24                                       goalMin,goalMax);
25
26           if (type == MIN && child[m].value < value ||
27               type == MAX && child[m].value > value) {
28               value = child[m].value;
29               bestMove = m;
30
31               // leave if goal is reached
32               if (type == MIN && value <= goalMin) return;
33               if (type == MAX && value >= goalMax) return;
34               // update goals
35               if (type == MIN) goalMax = value;
36               if (type == MAX) goalMin = value;
37     }  }  } // if, while, Konstruktor
```

"GameTreeNodeD.java"

Implementierung – SpielbaumD

```
1     private GameTreeNode(Position p,
2                          int goalMin, int goalMax) {
3         nodeCount++; pos = p; type = p.getPlToMv();
4         if (p.won() == -type) { value = -type; return; }
5         Iterator<Integer> moves = p.getMoves();
6         if (!moves.hasNext()) { value = NONE; return; }
7         int m;
8         if ((m=p.forcedWin(type)) != -1) {
9             bestMove = m;
10            value = type;
11            return;
12        }
13        if ((m=p.forcedWin(-type)) != -1) {
14            bestMove = m;
15            child[m] = new GameTreeNode(p.makeMove(m),
16                                        goalMin, goalMax);
17            value = child[m].value;
18            return;
19        }
```

"GameTreeNodeD.java"

Output – Variante D

```
generate tree... (2914 nodes)
x..
...
...
generate tree... (271 nodes)
x..
.o.
...
generate tree... (106 nodes)
xx.
.o.
...
generate tree... (9 nodes)
xxo
.o.
...
generate tree... (8 nodes)
xxo
.o.
x..
generate tree... (7 nodes)
xxo
oo.
x..
generate tree... (6 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

Implementierung – SpielbaumD

```
20        value = -2*type;
21        while (moves.hasNext()) {
22           m = moves.next();
23           child[m] = new GameTreeNode(p.makeMove(m),
24                                       goalMin,goalMax);
25
26           if (type == MIN && child[m].value < value ||
27               type == MAX && child[m].value > value) {
28               value = child[m].value;
29               bestMove = m;
30
31               // leave if goal is reached
32               if (type == MIN && value <= goalMin) return;
33               if (type == MAX && value >= goalMax) return;
34               // update goals
35               if (type == MIN) goalMax = value;
36               if (type == MAX) goalMin = value;
37    }  }  } // if, while, Konstruktor
```

"GameTreeNodeD.java"

Effizienz

Was könnte man noch tun?

- Eröffnungen; für die initialen Konfigurationen den besten Antwortzug speichern.
- Ausnutzen von Zugumstellungen. Überprüfen ob man die aktuelle Stellung schon irgendwo im Spielbaum gesehen hat (Hashtabelle).
- Ausnutzen von Symmetrien.

Aber für Tic-Tac-Toe wäre dieses wohl overkill...

Für komplexe Spiele wie Schach oder Go ist eine exakte Auswertung des Spielbaums völlig illusorisch...

Output – Variante D

```
generate tree... (2914 nodes)
x..
...
...
generate tree... (271 nodes)
x..
.o.
...
generate tree... (106 nodes)
xx.
.o.
...
generate tree... (9 nodes)
xxo
.o.
...
generate tree... (8 nodes)
xxo
.o.
x..
```

```
generate tree... (7 nodes)
xxo
oo.
x..
generate tree... (6 nodes)
xxo
oox
x..
generate tree... (5 nodes)
xxo
oox
xo.
generate tree... (2 nodes)
xxo
oox
xox
```

GUI: Model – View – Controller

Modell (Model):
Repräsentiert das Spiel, den aktuellen Spielzustand, und die Spiellogik.

Ansicht (View)
Die externe graphische(?) Benutzeroberfläche mit der die Benutzerin interagiert.

Steuerung (Controller)
Kontrollschicht, die Aktionen der Nutzerin and die Spiellogik weiterleitet, und Reaktionen sichtbar macht.

Typisch für viele interaktive Systeme. Es gibt viele Varianten (Model-View-Presenter, Model-View-Adapter, etc.).

Effizienz

Was könnte man noch tun?

- Eröffnungen; für die initialen Konfigurationen den besten Antwortzug speichern.
- Ausnutzen von Zugumstellungen. Überprüfen ob man die aktuelle Stellung schon irgendwo im Spielbaum gesehen hat (Hashtabelle).
- Ausnutzen von Symmetrien.

Aber für Tic-Tac-Toe wäre dieses wohl overkill...

Für komplexe Spiele wie Schach oder Go ist eine exakte Auswertung des Spielbaums völlig illusorisch...

GUI: Model – View – Controller

Modell (Model):
Repräsentiert das Spiel, den aktuellen Spielzustand, und die Spiellogik.

Ansicht (View)
Die externe graphische(?) Benutzeroberfläche mit der die Benutzerin interagiert.

Steuerung (Controller)
Kontrollschicht, die Aktionen der Nutzerin and die Spiellogik weiterleitet, und Reaktionen sichtbar macht.

Typisch für viele interaktive Systeme. Es gibt viele Varianten (Model-View-Presenter, Model-View-Adapter, etc.).

GUI: Model – View – Controller

- Es gibt viele solcher Standardvorgehensweisen, für das Strukturieren, bzw. Schreiben von großen Programmen (Design Patterns, ↑Softwaretechnik).
- Es gibt auch Anti Patterns, d.h., Dinge, die man normalerweise nicht tun sollte (die aber trotzdem häufig vorkommen).

GUI: Model – View – Controller

- Es gibt viele solcher Standardvorgehensweisen, für das Strukturieren, bzw. Schreiben von großen Programmen (Design Patterns, ↑Softwaretechnik).
- Es gibt auch Anti Patterns, d.h., Dinge, die man normalerweise nicht tun sollte (die aber trotzdem häufig vorkommen).

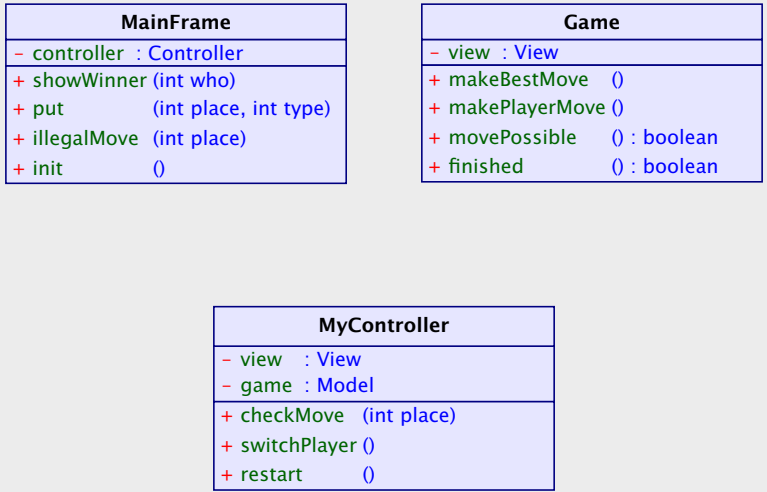
TicTacToe – GUI

MainFrame
- controller : Controller
+ showWinner (int who)
+ put (int place, int type)
+ illegalMove (int place)
+ init ()

Game
- view : View
+ makeBestMove ()
+ makePlayerMove ()
+ movePossible () : boolean
+ finished () : boolean

MyController
- view : View
- game : Model
+ checkMove (int place)
+ switchPlayer ()
+ restart ()

TicTacToe – GUI



View – Attributes

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import javax.swing.border.*;
5  import static javax.swing.SwingUtilities.*;
6
7
8  public class MainFrame extends JFrame implements
9                                       PlayConstants,View {
10     private Controller controller;
11     private JDialog dia;
12     private JPanel  arena;
13     private JPanel  side;
14     private Container c;
```

"MainFrame.java"

View – Attributes

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import javax.swing.border.*;
5  import static javax.swing.SwingUtilities.*;
6
7
8  public class MainFrame extends JFrame implements
9                                          PlayConstants,View {
10     private Controller controller;
11     private JDialog dia;
12     private JPanel  arena;
13     private JPanel  side;
14     private Container c;
```

"MainFrame.java"

View

```
16     public MainFrame(Controller con) {
17         controller = con;
18         setupUI();
19         init();
20     }
21     private void setupUI() {
22         c = getContentPane();
23         setLocation(100,100);
24         c.setLayout(new FlowLayout());
25         setVisible(true);
26     }
```

"MainFrame.java"

View – Interfacemethoden

```
27    public void init() { invokeLater(()->{
28        c.removeAll();
29        arena = new JPanel();
30        arena.setBackground(Color.BLUE);
31        arena.setBorder(new LineBorder(Color.BLUE, 5));
32        arena.setLayout(new GridLayout(3,3,5,5));
33        arena.setPreferredSize(new Dimension(600,600));
34        for (int i=0; i<9;i++) {
35            MyButton b = new MyButton(i);
36            b.addActionListener( this::buttonAction );
37            arena.add(b); }
38        side = new JPanel();
39        side.setPreferredSize(new Dimension(200,600));
40        side.setLayout(new BorderLayout());
41        JButton b = new JButton("switch sides");
42        b.addActionListener( this::switchAction );
43        side.add(b,BorderLayout.CENTER);
44        c.add(arena);
45        c.add(side);
46        pack();  });}
```

View

```
16    public MainFrame(Controller con) {
17        controller = con;
18        setupUI();
19        init();
20    }
21    private void setupUI() {
22        c = getContentPane();
23        setLocation(100,100);
24        c.setLayout(new FlowLayout());
25        setVisible(true);
26    }
```

"MainFrame.java"

View – Interfacemethoden

```
47    public void put(int place, int type) {
48        invokeLater(()->{
49            JPanel canvas;
50            if (type == MIN) canvas = new Cross();
51            else canvas = new Circle();
52            arena.remove(place);
53            arena.add(canvas, place);
54            revalidate();
55            repaint();
56        });
57    }
58    public void illegalMove(int place) {
59        System.out.println("Illegal move: "+place);
60    }
```

"MainFrame.java"

View – Interfacemethoden

```
27    public void init() { invokeLater(()->{
28        c.removeAll();
29        arena = new JPanel();
30        arena.setBackground(Color.BLUE);
31        arena.setBorder(new LineBorder(Color.BLUE, 5));
32        arena.setLayout(new GridLayout(3,3,5,5));
33        arena.setPreferredSize(new Dimension(600,600));
34        for (int i=0; i<9;i++) {
35            MyButton b = new MyButton(i);
36            b.addActionListener( this::buttonAction );
37            arena.add(b); }
38        side = new JPanel();
39        side.setPreferredSize(new Dimension(200,600));
40        side.setLayout(new BorderLayout());
41        JButton b = new JButton("switch sides");
42        b.addActionListener( this::switchAction );
43        side.add(b,BorderLayout.CENTER);
44        c.add(arena);
45        c.add(side);
46        pack();  });}
```

View – Interfacemethoden

```
61    public void showWinner(int who) {
62        String str = "";
63        switch(who) {
64        case -1: str = "Kreuz gewinnt!"; break;
65        case  0: str = "Unentschieden!"; break;
66        case  1: str = "Kreis gewinnt!"; break;
67        }
68        final String s = str;
69        invokeLater(()->dia = new MyDialog(this,s));
70    }
```

"MainFrame.java"

View – Interfacemethoden

```
47    public void put(int place, int type) {
48        invokeLater(()->{
49            JPanel canvas;
50            if (type == MIN) canvas = new Cross();
51            else canvas = new Circle();
52            arena.remove(place);
53            arena.add(canvas, place);
54            revalidate();
55            repaint();
56        });
57    }
58    public void illegalMove(int place) {
59        System.out.println("Illegal move: "+place);
60    }
```

"MainFrame.java"

View – ActionListener

```
72      public void switchAction(ActionEvent e) {
73          controller.switchPlayer();
74      }
75      public void buttonAction(ActionEvent e) {
76          MyButton button = (MyButton) e.getSource();
77          int place = button.getNumber();
78          controller.checkMove(place);
79      }
80      public void dialogAction(ActionEvent e) {
81          JButton b = (JButton) e.getSource();
82          if (e.getActionCommand() == "kill") {
83              System.exit(0);
84          } else {
85              controller.restart();
86              dia.dispose();
87          }
88      }
```

"MainFrame.java"

View – Interfacemethoden

```
61      public void showWinner(int who) {
62          String str = "";
63          switch(who) {
64          case -1: str = "Kreuz gewinnt!"; break;
65          case  0: str = "Unentschieden!"; break;
66          case  1: str = "Kreis gewinnt!"; break;
67          }
68          final String s = str;
69          invokeLater(()->dia = new MyDialog(this,s));
70      }
```

"MainFrame.java"

Controller – Attributes

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class MyController implements PlayConstants,
5                                       Controller {
6      private Model  game;
7      private View   view;
8      public void setup(Model m, View v) {
9          game = m; view = v;
10     }
```

"MyController.java"

View – ActionListener

```
72     public void switchAction(ActionEvent e) {
73         controller.switchPlayer();
74     }
75     public void buttonAction(ActionEvent e) {
76         MyButton button = (MyButton) e.getSource();
77         int place = button.getNumber();
78         controller.checkMove(place);
79     }
80     public void dialogAction(ActionEvent e) {
81         JButton b = (JButton) e.getSource();
82         if (e.getActionCommand() == "kill") {
83             System.exit(0);
84         } else {
85             controller.restart();
86             dia.dispose();
87         }
88     }
```

"MainFrame.java"

Controller – Methoden

```
11      public void checkMove(int place) {
12          if (game.movePossible(place)) {
13              game.makePlayerMove(place);
14          }
15          else view.illegalMove(place);
16      }
17      public void switchPlayer() {
18          if (game.finished()) return;
19          game.makeBestMove();
20      }
21      public void restart() {
22          view.init();
23          game = new Game(view);
24  }   }
```

"MyController.java"

Controller – Attributes

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class MyController implements PlayConstants,
5                                       Controller {
6      private Model  game;
7      private View   view;
8      public void setup(Model m, View v) {
9          game = m; view = v;
10     }
```

"MyController.java"

Main

```
1     public static void main(String[] args) {
2         invokeLater(()->{
3             Controller c = new MyController();
4             View  v = new MainFrame(c);
5             Model m = new Game(v);
6             c.setup(m,v);
7         });
8     }
```

"MainFrame.java"

Controller – Methoden

```
11    public void checkMove(int place) {
12        if (game.movePossible(place)) {
13            game.makePlayerMove(place);
14        }
15        else view.illegalMove(place);
16    }
17    public void switchPlayer() {
18        if (game.finished()) return;
19        game.makeBestMove();
20    }
21    public void restart() {
22        view.init();
23        game = new Game(view);
24 }  }
```

"MyController.java"

Diskussion

Was ist hier falsch?

Was passiert wenn wir einen sehr grossen Spielbaum berechnen?

Main

```
1    public static void main(String[] args) {
2        invokeLater(()->{
3            Controller c = new MyController();
4            View  v = new MainFrame(c);
5            Model m = new Game(v);
6            c.setup(m,v);
7        });
8    }
```

"MainFrame.java"

Diskussion

Was ist hier falsch?

Was passiert wenn wir einen sehr grossen Spielbaum berechnen?

Main

```
1    public static void main(String[] args) {
2        invokeLater(()->{
3            Controller c = new MyController();
4            View  v = new MainFrame(c);
5            Model m = new Game(v);
6            c.setup(m,v);
7        });
8    }
```

"MainFrame.java"

Main

```
1     private void initTree() {
2         try {Thread.sleep(4000);}
3         catch(InterruptedException e) {}
4         g.nodeCount = 0;
5         g = new GameTreeNode(p);
6         System.out.println("generate tree... (" +
7                                 g.nodeCount + " nodes)");
8     }
```

"GameNew.java"

Die GUI reagiert nicht mehr, da wir die gesamte Berechnung im Event Dispatch Thread ausführen.

Diskussion

Was ist hier falsch?

Was passiert wenn wir einen sehr grossen Spielbaum berechnen?

View – ActionListener

```
72  public void switchAction(ActionEvent e) {
73      ctrl.exec(()->ctrl.switchPlayer());
74  }
75  public void buttonAction(ActionEvent e) {
76      MyButton button = (MyButton) e.getSource();
77      int place = button.getNumber();
78      ctrl.exec(()->ctrl.checkMove(place));
79  }
80  public void dialogAction(ActionEvent e) {
81      JButton b = (JButton) e.getSource();
82      if (e.getActionCommand() == "kill") {
83          System.exit(0);
84      } else {
85          ctrl.exec(()->ctrl.restart());
86          dia.dispose();
87      }
88  }
```

"MainFrameNew.java"

Main

```
1     private void initTree() {
2         try {Thread.sleep(4000);}
3         catch(InterruptedException e) {}
4         g.nodeCount = 0;
5         g = new GameTreeNode(p);
6         System.out.println("generate tree... (" +
7                                 g.nodeCount + " nodes)");
8     }
```

"GameNew.java"

Die GUI reagiert nicht mehr, da wir die gesamte Berechnung im Event Dispatch Thread ausführen.

Controller – Attributes

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import java.util.concurrent.locks.*;
4
5  public class MyController extends Thread
6                             implements PlayConstants,
7                                        Controller {
8      private Model  game;
9      private View   view;
10     final Lock      lock = new ReentrantLock();
11     final Condition cond = lock.newCondition();
```

"MyControllerNew.java"

View – ActionListener

```
72   public void switchAction(ActionEvent e) {
73       ctrl.exec(()->ctrl.switchPlayer());
74   }
75   public void buttonAction(ActionEvent e) {
76       MyButton button = (MyButton) e.getSource();
77       int place = button.getNumber();
78       ctrl.exec(()->ctrl.checkMove(place));
79   }
80   public void dialogAction(ActionEvent e) {
81       JButton b = (JButton) e.getSource();
82       if (e.getActionCommand() == "kill") {
83           System.exit(0);
84       } else {
85           ctrl.exec(()->ctrl.restart());
86           dia.dispose();
87       }
88   }
```

"MainFrameNew.java"

Controller – Methoden

```
13      Runnable r = null;
14      public void exec(Runnable r) {
15          if (lock.tryLock()) {
16              this.r = r;
17              cond.signal();
18              lock.unlock();
19          }
20      }
21      public void run() {
22          lock.lock(); try {
23          while (true) {
24              while (r == null)
25                  cond.await();
26              r.run();
27              r = null;
28          }}
29          catch (InterruptedException e) {}
30          finally { lock.unlock(); }
31      }
```

Controller – Attributes

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import java.util.concurrent.locks.*;
4
5  public class MyController extends Thread
6                             implements PlayConstants,
7                                        Controller {
8      private Model  game;
9      private View   view;
10     final Lock      lock = new ReentrantLock();
11     final Condition cond = lock.newCondition();
```

"MyControllerNew.java"

Controller – Methoden

```
32      public void setup(Model m, View v) {
33          game = m; view = v;
34          start();
35      }
```

Controller – Methoden

```
13      Runnable r = null;
14      public void exec(Runnable r) {
15          if (lock.tryLock()) {
16              this.r = r;
17              cond.signal();
18              lock.unlock();
19          }
20      }
21      public void run() {
22          lock.lock(); try {
23          while (true) {
24              while (r == null)
25                  cond.await();
26              r.run();
27              r = null;
28          }}
29          catch (InterruptedException e) {}
30          finally { lock.unlock(); }
31      }
```

19 Collections Framework

Collection = Containterklasse, die andere Objekte enthält.

Inhalte:

- Schnittstellen
- Implementierungen
- Algorithmen

Vorteile:

- Einheitlicher Zugriff auf Containerobjekte.
- Abstraktion von den Implementierungsdetails.
- Effiziente Standardimplementierungen.
- Durch Einhalten vorgegebener APIs können unabhängige Containerklassen zusammenarbeiten.

19 Collections Framework

Collection = Containterklasse, die andere Objekte enthält.

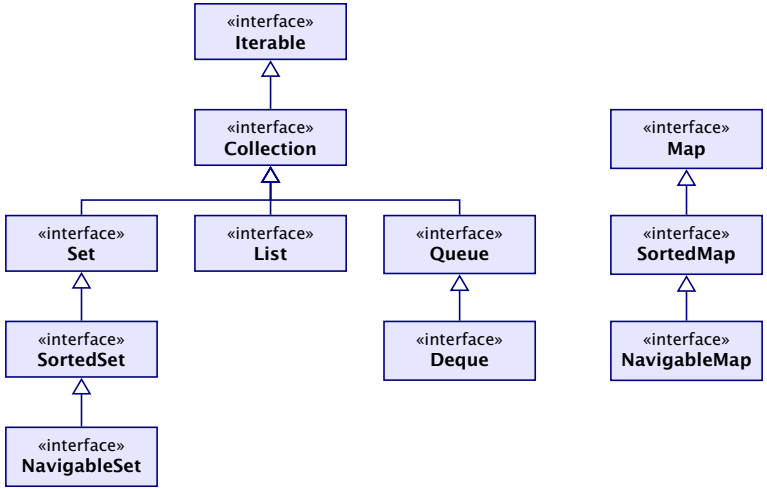
Inhalte:

- Schnittstellen
- Implementierungen
- Algorithmen

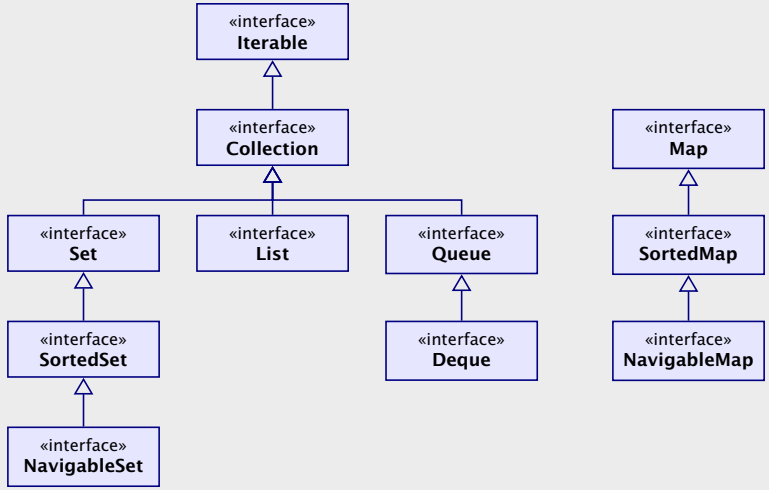
Vorteile:

- Einheitlicher Zugriff auf Containerobjekte.
- Abstraktion von den Implementierungsdetails.
- Effiziente Standardimplementierungen.
- Durch Einhalten vorgegebener APIs können unabhängige Containerklassen zusammenarbeiten.

Interfaces – Maps and Collections



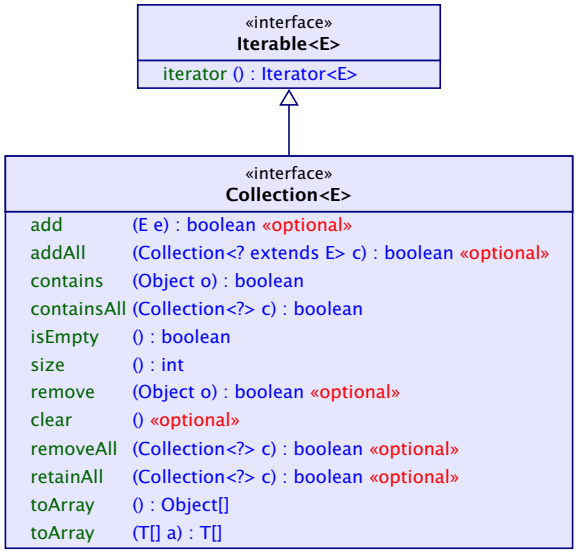
Interfaces – Maps and Collections



Überblick

- **Collection.** Allgemeine Containerschnittstelle; das Framework bietet keine konkrete Implementierung;
- **Set.** Ein Container, der keine Duplikate enthält.
- **List.** Ein geordneter Container; man kann Elemente an vorgegebenen Positionen einfügen; Duplikate sind erlaubt.
- **Queue.** Ein Container, der zusätzliche Einfüge, und Abfragemöglichkeiten bietet; realisiert (spezielle) Ordnung der Elemente (z.B. `FIFO`, `LIFO`, `PriorityQueue`)
- **Map** Assoziativer Speicher. Bildet Schlüssel auf Werte ab. Schlüssel können nicht doppelt vorkommen.

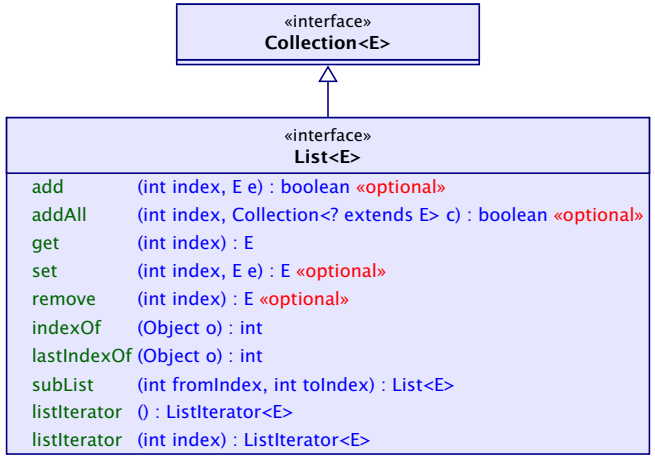
Collection Interface



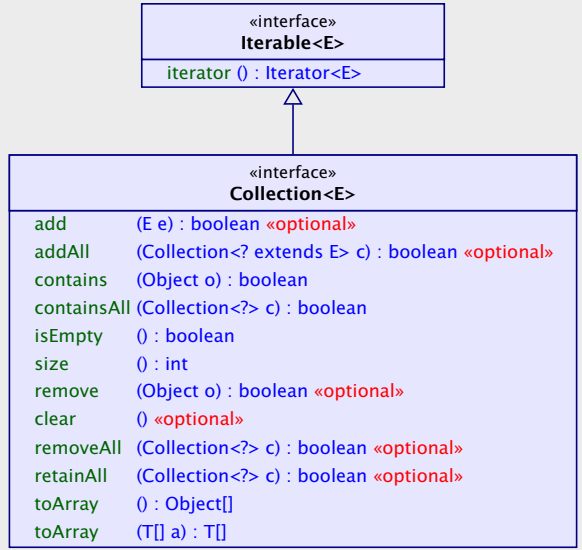
Überblick

- **Collection.** Allgemeine Containerschnittstelle; das Framework bietet keine konkrete Implementierung;
- **Set.** Ein Container, der keine Duplikate enthält.
- **List.** Ein geordneter Container; man kann Elemente an vorgegebenen Positionen einfügen; Duplikate sind erlaubt.
- **Queue.** Ein Container, der zusätzliche Einfüge, und Abfragemöglichkeiten bietet; realisiert (spezielle) Ordnung der Elemente (z.B. `FIFO`, `LIFO`, `PriorityQueue`)
- **Map** Assoziativer Speicher. Bildet Schlüssel auf Werte ab. Schlüssel können nicht doppelt vorkommen.

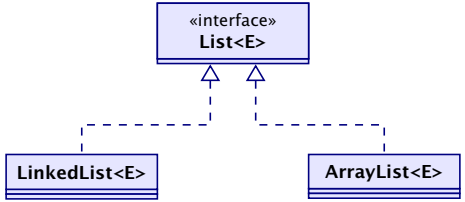
List Interface



Collection Interface



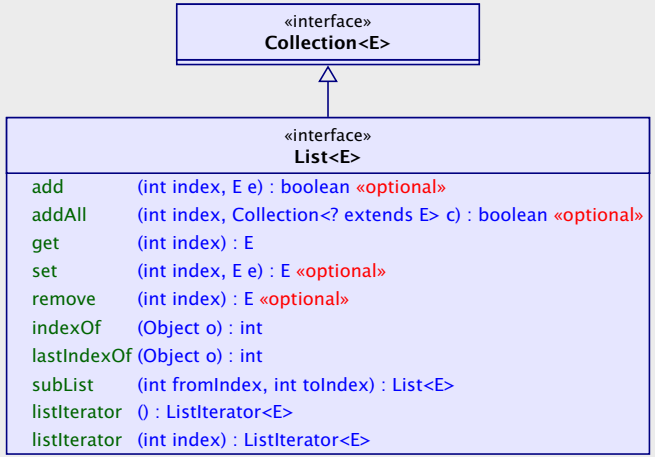
List – Implementierungen



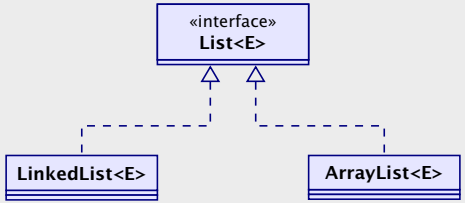
Laufzeiten:

	add	*remove*	*get*	*contains*
ArrayList	$O(1)$	$O(n)$	$O(1)$	$O(n)$
LinkedList	$O(1)$	$O(1)$	$O(n)$	$O(n)$

List Interface



List – Implementierungen



Laufzeiten:

	add	*remove*	*get*	*contains*
ArrayList	$O(1)$	$O(n)$	$O(1)$	$O(n)$
LinkedList	$O(1)$	$O(1)$	$O(n)$	$O(n)$

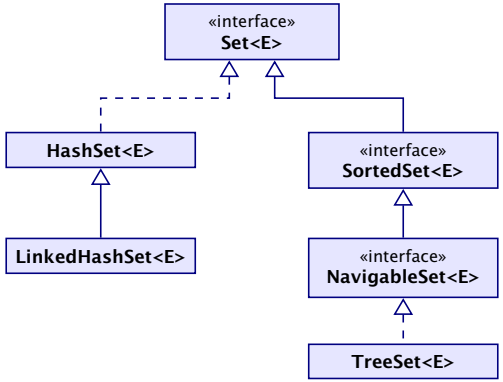
Set Interface(s)

Das Set-interface enthält nur Methoden aus dem Collection-Interface. Diese haben aber teilweise eine andere Bedeutung, da Duplikate nicht erlaubt sind.

Zusätzliche Methoden von SortedSet:

«interface» SortedSet<E>	
first	() : E
last	() : E
headSet	(E toElement) : SortedSet<E>
tailSet	(E fromElement) : SortedSet<E>
subSet	(E fromElement, E toElement) : SortedSet<E>
comparator	() : Comparator<? super E>

Set – Implementierungen



- `TreeSet<E>(Comparator<E> c)` erzeugt eine sortierte Menge, in dem Elemente gemäß `c` sortiert sind.

Set Interface(s)

Das `Set`-interface enthält nur Methoden aus dem `Collection`-Interface. Diese haben aber teilweise eine andere Bedeutung, da Duplikate nicht erlaubt sind.

Zusätzliche Methoden von `SortedSet`:

«interface» SortedSet<E>	
first	() : E
last	() : E
headSet	(E toElement) : SortedSet<E>
tailSet	(E fromElement) : SortedSet<E>
subSet	(E fromElement, E toElement) : SortedSet<E>
comparator	() : Comparator<? super E>

Set – Laufzeiten

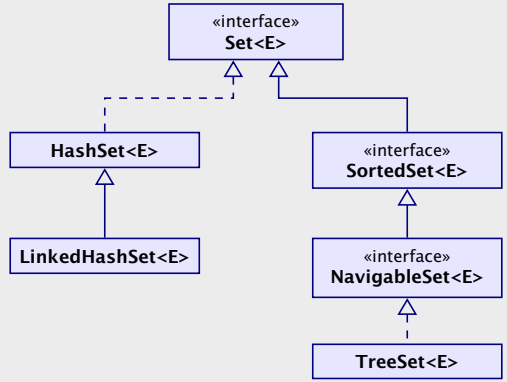
Laufzeiten:

	add	*contains*	*next*
`HashSet`	$O(1)$	$O(1)$	$O(h/n)$
`LinkedHashSet`	$O(1)$	$O(1)$	$O(1)$
`TreeSet`	$O(\log n)$	$O(\log n)$	$O(\log n)$

h ist die Anzahl der Buckets in der HashSet-Implementierung.

Die Laufzeiten für die hashbasierten Verfahren setzen eine gute `hashCode()`-Funktion voraus.

Set – Implementierungen



- `TreeSet<E>(Comparator<E> c)` erzeugt eine sortierte Menge, in dem Elemente gemäß `c` sortiert sind.

Set – Laufzeiten

Laufzeiten:

	add	*contains*	*next*
`HashSet`	$O(1)$	$O(1)$	$O(h/n)$
`LinkedHashSet`	$O(1)$	$O(1)$	$O(1)$
`TreeSet`	$O(\log n)$	$O(\log n)$	$O(\log n)$

h ist die Anzahl der Buckets in der HashSet-Implementierung.

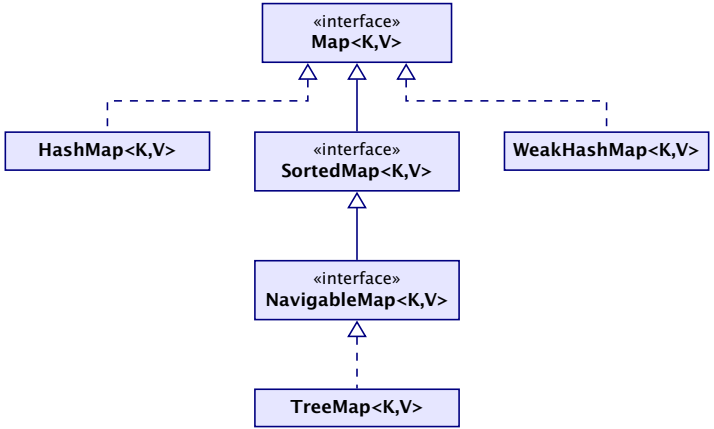
Die Laufzeiten für die hashbasierten Verfahren setzen eine gute `hashCode()`-Funktion voraus.

Map Interface

«interface» Map<K,V>	
put	(K key, V value) : V «optional»
putAll	(Map<? extends K,? extends V> m) «optional»
remove	(Object o) : V «optional»
clear	() «optional»
containsKey	(Object o) : boolean
containsValue	(Object o) : boolean
isEmpty	() : boolean
size	() : int
get	(K key) : V
entrySet	() : Set<Map.Entry<K,V>>
keySet	() : Set<K>
values	() : Collection<V>

- Schlüssel werden auf Werte abgebildet;
- kein Schlüssel kommt doppelt vor; Werte eventuell schon
- in einer `SortedMap` sind die Schlüssel sortiert

Map – Implementierungen

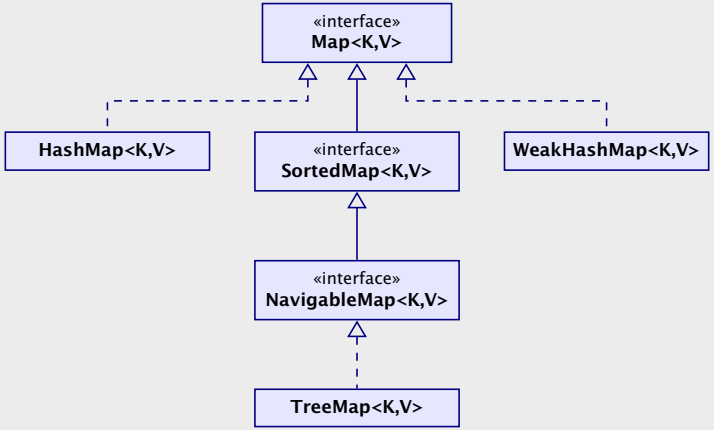


Map Interface

«interface» Map<K,V>	
put	(K key, V value) : V «optional»
putAll	(Map<? extends K,? extends V> m) «optional»
remove	(Object o) : V «optional»
clear	() «optional»
containsKey	(Object o) : boolean
containsValue	(Object o) : boolean
isEmpty	() : boolean
size	() : int
get	(K key) : V
entrySet	() : Set<Map.Entry<K,V>>
keySet	() : Set<K>
values	() : Collection<V>

- Schlüssel werden auf Werte abgebildet;
- kein Schlüssel kommt doppelt vor; Werte eventuell schon
- in einer `SortedMap` sind die Schlüssel sortiert

Map – Implementierungen



Map – Laufzeiten

Laufzeiten:

	get	*containsKey*	*next*
HashMap	$O(1)$	$O(1)$	$O(h/n)$
WeakHashMap	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$

h ist die Anzahl der Buckets in der HashSet-Implementierung.

Die Laufzeiten für die hashbasierten Verfahren setzen eine gute hashCode()-Funktion voraus.

Map – Laufzeiten

Laufzeiten:

	get	*containsKey*	*next*
HashMap	$O(1)$	$O(1)$	$O(h/n)$
WeakHashMap	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$

h ist die Anzahl der Buckets in der HashSet-Implementierung.

Die Laufzeiten für die hashbasierten Verfahren setzen eine gute hashCode()-Funktion voraus.

Collections – Tricks

- Löschen, der Elemente $[100,\ldots,200)$ in einer Liste:

```
list.subList(100,200).clear();
```

- Entfernen von Duplikaten aus einer Collection:

```
Collection<Type> c = ...;
Collection<Type> noDups = new HashSet<Type>(c);
```

- Löschen aller Strings, die mit 'f' anfangen:

```
SortedSet<String> dictionary = ...;
dictionary.subSet("f", "g").clear();
```

Collections – Tricks

- Löschen, der Elemente $[100,\ldots,200)$ in einer Liste:

```
list.subList(100,200).clear();
```

- Entfernen von Duplikaten aus einer `Collection`:

```
Collection<Type> c = ...;
Collection<Type> noDups = new HashSet<Type>(c);
```

- Löschen aller Strings, die mit `'f'` anfangen:

```
SortedSet<String> dictionary = ...;
dictionary.subSet("f", "g").clear();
```


Arrays – asList

Die Funktion `asList(T[] a)` gibt ein Listenview auf ein Array zurück. Damit können Funktionen, die eine `Collection` erwarten auch mit einem Array aufgerufen werden:

```
1 Integer[] arr = {3,7,12,-5};
2 Collections.sort(Arrays.asList(arr));
3 for (int i=0; i<arr.length; i++) {
4     System.out.println(arr[i]);
5 }
```

- die resultierende Liste ist in der Größe fixiert; `remove()`, `add()` werfen Exceptions;
- mit `set()` kann man die Liste aber verändern.

Arrays – asList

Die Funktion `asList(T[] a)` gibt ein Listenview auf ein Array zurück. Damit können Funktionen, die eine `Collection` erwarten auch mit einem Array aufgerufen werden:

```
1 Integer[] arr = {3,7,12,-5};
2 Collections.sort(Arrays.asList(arr));
3 for (int i=0; i<arr.length; i++) {
4     System.out.println(arr[i]);
5 }
```

- die resultierende Liste ist in der Größe fixiert; `remove()`, `add()` werfen Exceptions;
- mit `set()` kann man die Liste aber verändern.

Eigene Kollektionen

- Zum Erstellen eigener Kollektionen erbt man von abstrakten Klassen, die einen Großteil der Implementierung bereitstellen (z.B. `removeAll()` über wiederholten Aufruf von `remove()` etc.)
- Dann werden nur einige Funktionen implementiert.
- Für zusätzliche Effizienz können auch weitere Funktionen überschrieben werden.

Eigene Kollektionen

- Zum Erstellen eigener Kollektionen erbt man von abstrakten Klassen, die einen Großteil der Implementierung bereitstellen (z.B. `removeAll()` über wiederholten Aufruf von `remove()` etc.)
- Dann werden nur einige Funktionen implementiert.
- Für zusätzliche Effizienz können auch weitere Funktionen überschrieben werden.

Eigene Kollektionen – Übersicht

- `AbstractCollection` benötigt `iterator` und `size`.
- `AbstractSet` benötigt `iterator` und `size`.
- `AbstractList` benötigt `get` und `size` und (optional) `set`, `remove`, `add`.
- `AbstractSequentialList` benötigt `listIterator` und `size`.
- `AbstractMap` benötigt `entrySet` (View); (optional) `put` falls veränderbar

Beispiel – asList

Idee:

- Speichere Arrayreferenz in Attribut lokaler Klasse.
- Übersetze Listenbefehle in entsprechende Arraybefehle.
- typisch für Adapterklassen

```
1 public static <T> List<T> asList(T[] a) {
2     return new MyArrayList<T>(a);
3 }
4 private static class MyArrayList<T> extends
5                                     AbstractList<T> {
6     private final T[] a;
7     MyArrayList(T[] array) {
8         a = array;
9     }
```


Eigene Kollektionen – Übersicht

- AbstractCollection benötigt iterator und size.
- AbstractSet benötigt iterator und size.
- AbstractList benötigt get und size und (optional) set, remove, add.
- AbstractSequentialList benötigt listIterator und size.
- AbstractMap benötigt entrySet (View); (optional) put falls veränderbar

Beispiel – asList

Idee:

- Speichere Arrayreferenz in Attribut lokaler Klasse.
- Übersetze Listenbefehle in entsprechende Arraybefehle.
- typisch für Adapterklassen

```
1  public static <T> List<T> asList(T[] a) {
2      return new MyArrayList<T>(a);
3  }
4  private static class MyArrayList<T> extends
5                                          AbstractList<T> {
6      private final T[] a;
7      MyArrayList(T[] array) {
8          a = array;
9      }
```


Beispiel – asList

```
10     public T get(int index) {
11         return a[index];
12     }
13     public T set(int index, T element) {
14         T oldValue = a[index];
15         a[index] = element;
16         return oldValue;
17     }
18     public int size() {
19         return a.length;
20     }
21 }
```

Iterieren über Kollektionen

Seit Java2:

```
1 List<String> names = Arrays.asList("Peter", "Max",
      "Moritz", "Lukas");
2 Iterator<String> i = names.iterator();
3 for (; i.hasNext(); name=i.next())
4     System.out.println(name);
```

Seit Java5:

```
1 for (String name : names) {
2     System.out.println(name);
3 }
```

Seit Java8:

```
1 names.forEach(n->System.out.println(n));
```

Streams

```
List<String> names =
    Arrays.asList("Moritz", "Lukas", "Max", "Peter");

names
    .stream()
    .filter(s -> s.startsWith("M"))
    .map(String::toLowerCase)
    .sorted()
    .forEach(System.out::println);
```

Iterieren über Kollektionen

Seit Java2:

```
List<String> names = Arrays.asList("Peter", "Max",
    "Moritz", "Lukas");
Iterator<String> i = names.iterator();
for (; i.hasNext(); name=i.next())
    System.out.println(name);
```

Seit Java5:

```
for (String name : names) {
    System.out.println(name);
}
```

Seit Java8:

```
names.forEach(n->System.out.println(n));
```

Stream Pipeline

Eine „source“

- Z.B.: `list.stream()` erzeugt Stream aus Kollektion `list`;

Mehrere „intermediate operations“

- `.filter()`, filtert Objekte aus, für die geg. Prädikat gilt
- `.map()`, wendet eine Funktion auf jedes Objekt an
- ...

Eine „terminal operation“

- `.forEach()` wendet Funktion auf jedes Element an
- `.reduce()` berechnet ein Ergebnis aus allen Werten
- `.collect()` kann z.B. Elemente in Kollektion speichern
- `.findFirst()`, liefert ein `Optional` mit erstem Element
- ...

Streams

```
1 List<String> names =
2     Arrays.asList("Moritz", "Lukas", "Max", "Peter");
3
4 names
5     .stream()
6     .filter(s -> s.startsWith("M"))
7     .map(String::toLowerCase)
8     .sorted()
9     .forEach(System.out::println);
```

Streams

```
1 int res =
2     Stream.of(7, 3, -5, 15)
3         .filter(j -> j>0)
4         .mapToInt(j->j)
5         .sum();
```


Stream Pipeline

Eine „source“

- Z.B.: `list.stream()` erzeugt Stream aus Kollektion `list`;

Mehrere „intermediate operations“

- `.filter()`, filtert Objekte aus, für die geg. Prädikat gilt
- `.map()`, wendet eine Funktion auf jedes Objekt an
- ...

Eine „terminal operation“

- `.forEach()` wendet Funktion auf jedes Element an
- `.reduce()` berechnet ein Ergebnis aus allen Werten
- `.collect()` kann z.B. Elemente in Kollektion speichern
- `.findFirst()`, liefert ein `Optional` mit erstem Element
- ...

Streams

```
1 List<String> filtered =
2     names // Kollektion mit Strings
3         .stream()
4         .filter(s -> s.startsWith("P"))
5         .collect(Collectors.toList());
```


Streams

```
1 int res =
2     Stream.of(7, 3, -5, 15)
3         .filter(j -> j>0)
4         .mapToInt(j->j)
5         .sum();
```

Streams

```
List<String> filtered =
    names // Kollektion mit Strings
        .stream()
        .filter(s -> s.startsWith("P"))
        .collect(Collectors.toList());
```


Streams

Eine Version von `.reduce()` erwartet eine (assoziative) Funktion mit zwei Argumenten:

```
Optional<Integer> min =
    Stream.of(7, 3, -5, 15)
        .reduce((x,y)-> x < y ? x : y);
```

Im Prinzip wird diese Funktion in beliebiger Reihenfolge auf die Elemente des Streams angewendet, bis man nur noch einen Wert hat.

Durch die Assoziativität erhält man immer das gleiche Ergebnis.